Google

Web Images Groups News Froogle Local more »

history of drag and drop technology Search Advanced Search Preferences

The "AND" operator is unnecessary -- we include all search terms by default. [details]

Web Results **1** - **10** of about **825,000** for **history of drag and drop technology**. (**0.34** seconds)

Tip: Save time by hitting the return key instead of clicking on "search"

Web Applications 1.0
Undo **history**; 4.3. **Drag** and **drop**. 4.3.1. **Drag**-and-**drop** processing model ...
What is required are extensions to these **technologies** to provide much-needed ...
whatwg.org/specs/web-apps/current-work/ - 101k - Jul 12, 2005 - Cached - Similar pages

Persistence, **Drag** and **Drop**, and More to Make Developers Smile ...
DHTML now supports full application-level **drag**-and-**drop** events. This means that you can **drag** between frames and **drop**, as well as to and from applications. ...
msdn.microsoft.com/library/ en-us/dhtmltechcol/dndhtml/IE5new.asp - 24k - Cached - Similar pages

A Short **History** of DragThing
... that I was working on my **drag** and **drop** based dock during the summer of 1994, ... 10 Killer Apple **Technologies** Of The Last 10 Years That DragThing Didn't ...
www.**drag**thing.com/english/tenyears.html - 27k - Cached - Similar pages

f11photo.com - Online Photo Processor
If you are using **drag** & **drop**, you must be on an IBM computer and have ...
Adding pictures couldn't be easier now with our new **drag** and **drop technology**. ...
www.f11photo.com/opp/front.aspx?p=support - 16k - Cached - Similar pages

Sagan **Technology** Metro :: View topic - Metro 6.3.1 pre-release
Sagan **Technology** Metro Forum Index, Sagan **Technology** Metro ... New: **Drag** and **Drop** has been vastly improved so that you can **drag** a MIDI or audio file to a ...
www.sagantech.com/metroforum/viewtopic.php?t=908 - 31k - Cached - Similar pages

Ferl - A-Level **History** German Propaganda: **Drag**-and-**drop** exercise
A **drag**-and-**drag** exercise designed for students of A-Level **History** who are taking modules on ... A-Level **History** German Propaganda: **Drag**-and-**drop** exercise ...
ferl.becta.org.uk/display.cfm?resID=8822& page=628&catID=172&printable=1 - 17k - Cached - Similar pages

Speed Download 3 | Version **history**
Built-in **history** for all downloads: keep track of all your downloaded files via ... Complete **drag** 'n' **drop** support: **drag** a URL onto SD or the SD floater to ...
www.yazsoft.com/**history**.html - 35k - Cached - Similar pages

History of the Apple Computer
windows, **drag**-and-**drop** file moveability, and plug-in-and-play compatibility, ...
This article by no means attempts to relate the lengthy **history** of ...
www.thescreamonline.com/**technology**/ apple**history**/apple**history**.html - 20k - Cached - Similar pages

Zyphoto - Print digital photos at local lab
... and then select to use our **drag** and **drop technology**. For the **drag** and **drop**, just highlight all the pictures you would like to add and then **drag** it over ...
www.zyphoto.com/support.asp - 15k - Cached - Similar pages

[PDF] 12931 IT consultants bro/2

File Format: PDF/Adobe Acrobat - View as HTML
Drag-and-**drop technology** for fast application screen design ... to maintain summary, detail and **history** information using **drag**-and-**drop** tools within Screen ...
www.tatainfotech.com/sostenuto/ Tata_Sostenuto_IT_Consultants_Brochure.pdf - Similar pages

Gooooooooogle ▶
Result Page: 1 2 3 4 5 6 7 8 9 10 **Next**

history of drag and drop technology Search

Search within results | Language Tools | Search Tips | Dissatisfied? Help us improve

Google Home - Advertising Programs - Business Solutions - About Google

Mac & Apple History

Macintosh History: 1994

Dan Knight

If you can't beat 'em, join 'em.

That seems to be the philosophy behind the Houdini card that turned a normal Quadra 610 into the dual-platform Quadra 610 DOS Compatible. At a keystroke, the user could move between the comfortable Mac environment and the dominant, sometimes required, Windows platform.

Also introduced in February 1994 were the LC 550 and LC 575, faster cousins of the all-in-one LC 520. The 550 has a 33 MHz 68030, while the 575 moved to Quadra level with a 33 MHz 68LC040.

But these were just evolutionary developments; the revolution was one month off.

Power To the People

March 1994 marked the beginning of the end for the Motorola 680X0 line of processors, at least as far as the Macintosh was concerned.

Three new models, the Power Mac 6100, 7100, and 8100, moved the Mac platform from CISC chips to RISC chips. But most amazing of all, Apple had so carefully engineered the Power Macs, and so closely worked with Motorola and IBM in designing the PowerPC chip, that the Power Macs tended to be more compatible with old software than the Quadra series had been when it was first introduced.

Apple's wizards had created an emulator which worked so well that even some programs written for the original 1984 Macintosh could run flawlessly, albeit far faster, on the new Macs with an entirely new processor.

Needless to say, Apple sold over a million Power Macs within one year of their introduction.

But Apple didn't immediately abandon the 680X0 chips. One reason was that there was no low power version of the PowerPC for use in laptops.

Portable Power

May saw the introduction of six 68040-based PowerBooks, the Duo 280 and 280c, and the PowerBook 520, 520c, 540, and 540c. The extra power compared with the 68030 made these very popular, as did the color screens on the 280c, 520c, and 540c.

The 500-series laptops, a.k.a. Blackbird, was one of Apple's finest PowerBook designs, and the last to come from the factory with a modem and ethernet port until the PowerBook G3 of 1997.

System 7.5

It seems like we've been using System 7.5 forever, but it was only introduced in June 1994. I can't even recall what was so new and cool about 7.5, only that we quickly standardized on it at work -- and still use it for everything from the IIsi up through most of our Power Macs.

Fortunately, a reader has contributed the following list:

1. Apple Menu Options with the hierarchical Apple Menu and the recently used items.
2. The Launcher became standard, not just for the Performa.
3. General Controls was enhanced, with the options to protect the System and Application folders, the shut down warning, whether to show the desktop when the Finder was hidden, whether to launch the Launcher automatically, and which window to display when you open and save a document.
4. Stickies.
5. New desktop patterns.
6. WindowShade.
7. A new face-plate for the AppleCD Audio Player.
8. Better Find File, with things like more search options per search, numbers of items to display, and so on.
9. Apple Guide. A lot more comprehensive than the older Finder Help.
10. PowerTalk for things like directories and e-mail services on the desktop.
11. QuickDraw GX.
12. The Documents folder could be created automatically by the General Controls.
13. Menu Bar clock, based on SuperClock.
14. Desktop printer icons. Not much help if you have only one printer.
15. Drag and Drop
16. MacTCP became standard.
17. The Finder is purportedly faster, and is scriptable.
18. FAT. One set of disks, all computers (68K and PowerPC), at least at that time.

Apple Adopts IDE

Apple had a reputation for innovation, and for sticking with their own standards. Hard drives were SCSI, plain and simple.

At least they were until June 1994, when Apple shipped the Quadra 630 (a.k.a. Performa 630, LC 630) with an IDE hard drive, the same kind of hard drive used in the Wintel world.

Put simply, IDE drives were a fair bit less expensive than SCSI drives and offered

Mac History
1984 | 1985 | 1986
1987 | 1988 | 1989
1990 | 1991 | 1992
1993 | 1994 | 1995
1996 | 1997 | 1998
1999 | 2000 | 2001
2002 | Next

reasonable performance for entry level computers -- and today their descendants offer incredible performance on the Power Mac G3 Pro.

But it took Mac users by surprise to see an IDE drive in a Macintosh. Some viewed it as selling out to the dark side.

A Cheaper PowerBook

The only PowerBook at home is my wife's PowerBook 150, purchased after the price fell below $1,000. Her machine has 4 MB of memory, a 120 MB hard drive, and a 33 MHz 68030 CPU, making it a nice field machine.

Even today Apple seems incapable of selling a PowerBook at the $1,000 mark.

One way Apple kept the 150's cost down was using an IDE drive, just like the Quadra 630.

Power Performa, Power DOS

In September, Apple put the Performa label on the Power Mac 6100. And in November, they shipped a Power Mac 6100 DOS Compatible, again making it easy for Mac users to work in the Windows world when they had to.

The Competition

The biggest computer story of the year wasn't the PowerPC. It was the flaw discovered in the Pentium CPU, which eventually led Intel to recall each and every Pentium on the market.

Personal Perspective

I worked as a book designer when the Power Macs first came out. My specialty was, and remains, academic books with footnotes, which we design in FrameMaker. It's an incredibly powerful program, but it was excruciatingly slow on a IIci. Things improved on the Quadra, where it was just bearable.

But on the Power Mac . . . why, on the Power Mac, FrameMaker could sing! Others may rave about Photoshop speed improvements or smoother gaming, but I know the power of the PowerPC when I ran the PowerPC native version of FrameMaker.

It went from being sluggish to being faster than Quark, the program our other designers used (and continue to use). Suddenly, FrameMaker wasn't just powerful, but it became an efficient tool. (Before that, is was the only tool, so we had to make do with its lackluster performance.)

I'm a late adopter of the PowerPC at home; I graduated from a Centris 610 to a Umax SuperMac J700 in June 1998, mostly because my web work for Low End Mac demanded a lot more horsepower than I had -- and the $800 close-out price on the SuperMac matched the cost of upgrading my Centris to twice its current performance.

Instead, for the same money I saw at least a tenfold performance increase.

I may never run Mac OS X on this machine, but I'm very satisfied with 8.1, 104MB of memory, a 2.1GB hard drive, and a fast (compared to my 1x CD-ROM) 8x CD-

ROM player. And it's nearly as fast as the 7600/200 I use at work, although that'll change in a few weeks when we drop in a G3 card.

Still, the reason I run Low End Mac is simply that few of us need the latest, greatest computer technology -- and when we do buy it, we feel outdated within six months. Instead, Low End Mac grew out of my philosophy of getting the most from what you have and learning to be content with less than state of the art equipment. ELEM

ABOUT
WHAT'S NEW
HISTORY
DOWNLOAD
REGISTER
SUPPORT
FORUMS

A Short History of DragThing - Ten Years Of Tidying Up Your Desktop

May 2005 marks the tenth anniversary of the original release of DragThing 1.0. Ten years is a long time to be working on one piece of software, so I thought it was only appropriate to celebrate the occasion!

In the beginning...

This page will be updated over the next ten days with various bits of trivia and previously undisclosed secrets about DragThing that you won't find anywhere else.

If you are new to DragThing, you can find out more information about it here, and download a copy of the latest DragThing 5.6 with Tiger support here to try it out.

It's okay, we'll wait for you to come back before we start.

- 10 Things You Didn't Know About DragThing
- 10 Year Timeline
- 10 Killer Apple Technologies Of The Last 10 Years That DragThing Didn't Support
- 10 Years Of Screenshots
- 10 Applications Used To Make DragThing
- 10 Comments

There have been thirty-nine different releases of DragThing so far - if you've used DragThing since the very beginning or if you have any questions, I'd like to hear fro you!

Send me a comment to the address below, or post on the DragThing Forums and I'l add it to this page.

admin@dragthing.com

Here's to the next ten years!

10 Things You Didn't Know About DragThing

1. The official story of DragThing's name is that I was working on my drag and drop based dock during the summer of 1994, and a friend of mine named Dai asked me something along the lines of "have you finished that drag thing yet? and the name stuck. However, since I have told that story so many times, I can't remember if it is actually true or not, and I suspect it isn't.

2. The name was originally intended to be a parody of the many "InterCaps" product names of the time, especially in the Apple world - AppleScript, QuickTime, MacWrite and so on. I'd never originally intended to make any money with DragThing, so it didn't matter - or so I thought - that it had a marketable name. However, DragThing 1.0 proved very popular and I was stu with it. Over the years, quite a few products appeared from other developers with the word "Thing" at the end which amused me to no end.

3. DragThing was originally freeware, or more accurately "coolware". The idea w if you liked it and used it, you were supposed to send me something cool. Mos commonly this involved T-shirts, but I got a lot of interesting stuff sent to me from all around the world. Some time later, I got an email from a guy at a ver large advertising company who explained that his finance department didn't understand the concept of coolware, and would it be okay if they just sent a large cheque instead? And so, capitalism was invented. A beneficial side effect of this change was that it somewhat alleviated the mounting T-shirt storage crisis - to the great delight of my significant other.

4. Early versions of DragThing used to expire a year after they were released, ar started warning people to download a new copy a month prior to that. I did th because I was fed up with people running ancient versions and asking about bugs that I had fixed years ago. Some people thought it was a cunning schem that I had planned years in advance to make people to upgrade to the shareware version, but it wasn't. Honest!

5. A certain utility software company once offered to buy the exclusive rights to DragThing for $10,000. I respectfully declined, and they bought out one of my similarly-named competitors instead. When DragThing became shareware son time later, it soon became clear that this had been the correct decision!

6. DragThing Lite, the simplified version of DragThing which shipped with DragThing 2, was originally written to be included with Mac OS 8.5 instead of the AppSwitcher. The deal with Apple fell apart quite quickly, but the code survived.

7. To enable the hidden debug mode in DragThing 2, open the Preferences windc and type on the arrow keys: up up down down up up up up.

This was the same code that enabled the debug mode in the game Sonic The Hedgehog 3 on the Sega Genesis/Megadrive.

8. The reason there wasn't much DragThing development between 1998 and 20(is because I was then part of the team working in secret on the Finder and Ma OS X Dock at Apple. I felt then that the Dock would ultimately replace DragThing. However, for a variety of reasons, I left Apple shortly after Aqua a the Dock were announced in January 2000, and very little of my code survive: in the current Dock. I moved back to Scotland, and work started on DragThin(soon after.

   Update: I'm reliably informed that *none* of my original code survives in the current Dock. Thanks John for rubbing that in :-)

9. DragThing 3.0 was an unreleased version of DragThing 2.9 that had been adapted to run on early developer previews of Mac OS X. Because of Aqua, I realised I would need to rewrite much of the user interface code, and so I started working on DragThing 4 instead.

10. DragThing 4 on Mac OS X contained the coolest easter egg in the world, but fewer than ten people have seen it.

    Update: Many people have asked me what it is. I would love to tell you, becat it was really cool. But there would be *repercussions.*

10 Year Timeline

Some notable events over the last ten years:

1994	Work starts on DragThing 1.0 in the summer. PowerPC based Macintoshes first introduced. System 7.5 released.
1995	DragThing 1.0 is first released to the public in May. Cyberdog and OpenDoc announced at WWDC. Windows 95 released.
1996	I start working for a well known computer company. Apple buys NeXT.
1997	DragThing 2.0, the first shareware version, is released in July.

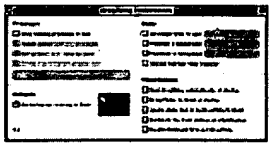
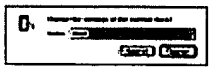
	Mac OS 8 released.
1998	The official dragthing.com site goes online in February. iMac introduced. Mac OS X announced.
1999	A quiet year for DragThing due to work on that *other* dock. iBooks and Airport introduced. Power Mac G4 ships.
2000	I stop working for a well known computer company. Aqua shown publicly for the first time.
2001	DragThing 4.0 for Mac OS X is released a few days before Mac OS X is. iPod revealed.
2002	DragThing 4.5 adds full support for Mac OS X 10.2 "Jaguar". LCD iMac ships with a bundled copy of PCalc.
2003	DragThing 5.0 adds full support for Mac OS X 10.3 "Panther". Safari introduced. iTunes Music Store opens its doors.
2004	DragThing 5.5 finally adds support for spring-loaded folders! 30" cinema display unveiled. iMac G5 ships.
2005	DragThing 5.6 adds full support for Mac OS X 10.4 "Tiger".

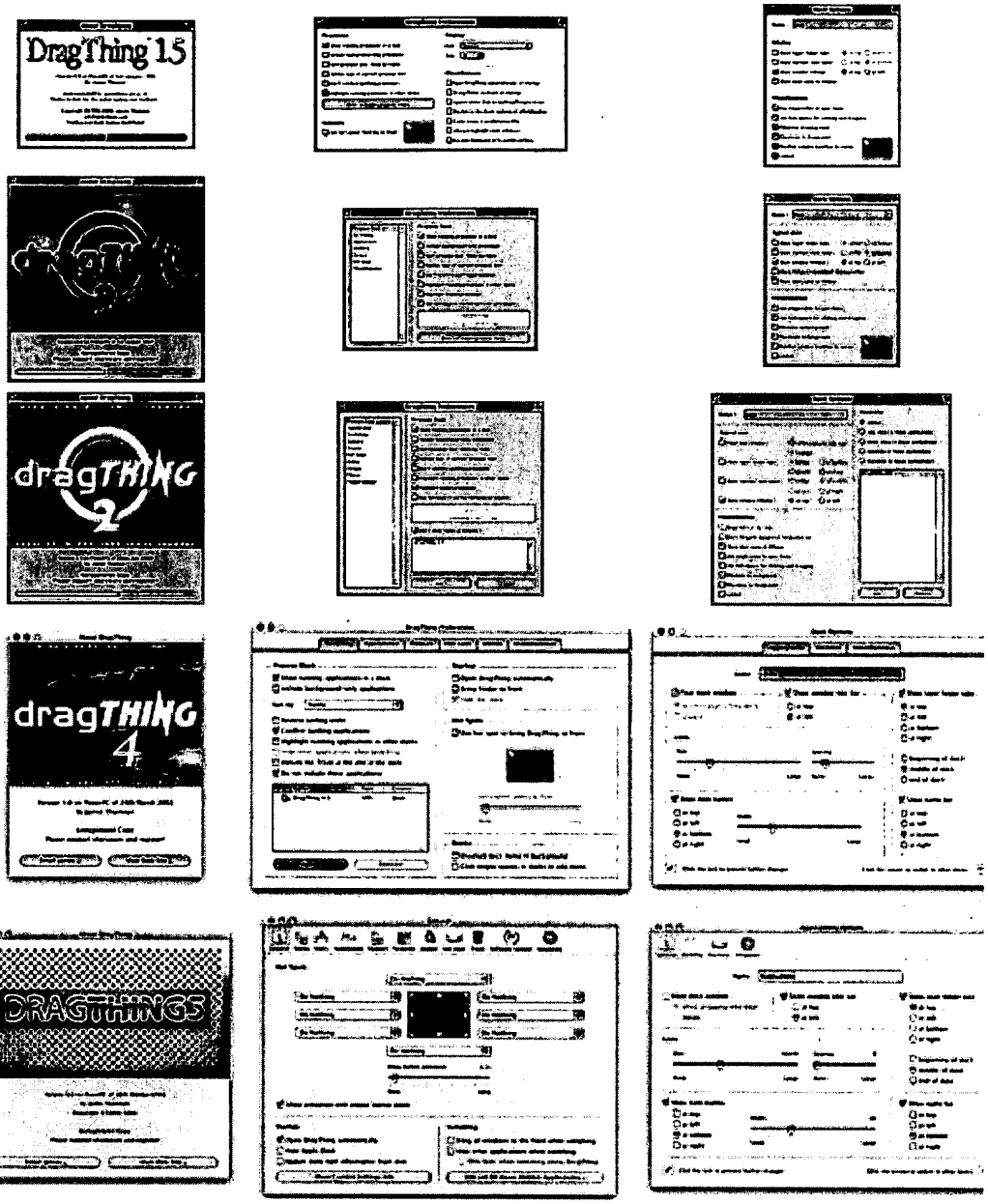
10 Killer Apple Technologies Of The Last 10 Years That DragThing Didn't Support

1. Copland
2. Rhapsody
3. OpenDoc
4. PowerTalk
5. QuickDraw GX
6. QuickDraw 3D
7. Publish & Subscribe
8. Newton OS
9. Dylan
10. Cocoa

10 Years Of Screenshots

Click the images below to see how things have evolved:

	About	Preferences	Dock Options
1995			
1996			



10 Applications Used To Make DragThing

1. Metrowerks Codewarrior - the development environment I've used for the last ten years
2. Apple Xcode - the development environment I'm likely to be using for the nex ten years, unfortunately
3. Bare Bones BBEdit - simply the best text editor
4. Adobe Photoshop - the best image editor, but certainly not simple
5. Resorcerer - at least it runs natively on Mac OS X!
6. ResEdit - the original. Still runs on Tiger
7. PowerGlot - so much nicer than AppleGlot
8. Iconographer - because everybody needs icons
9. Installer VISE / FileStorm - for installers past and disk images present
10. PCalc - the only software older than DragThing

10 Comments

1. From Cletus Waldman at ResExcellence:

   You've read this many times I am sure about various pieces of software. "It is the first item i install when I load a new system." But that is what is done on machines with this little jewel. There is nothing I have ever found to replace DragThing. Nothing! Sure there are alternatives out there in third party space None with the feature set of DT. I happen to be a point and click person and a keyboard shortcut guy. So this fits my work style perfectly and flawlessly.Th amazing thing about it however, is that it is such a dynamic and customizable program that one can literally have the best of both worlds— both point and click and keyboarding. It seems it is always literally one step ahead of the curve. I have been fortunate to be on the beta team also, so I know the care and work James puts in on this program. James you have one great little program there. I congratulate you, and wish you continued success in this wo of one shot shareware wonders.

2. From Guillaume Gete:

   *I began starting working with DragThing since 1995, or something like that. A to say that DT is one of the most stable pieces of software I've ever seen, eve at the time of Classic Mac OS, is an understatement. Funny to see that one of my oldest files on my Mac is still the DragThing preferences file from 1996... That could be a proof by itself. The fact is, from all the applications and utilitie I have tested for years on the Mac platform, DragThing is maybe the one with the most respect for the Mac itself, whether it is in its programming execution its carefulness of using the best Mac OS technologies in the best possible way. (do you know so many applications that on the 29th of April could use Quartz Extreme effects, AppleScript and support Finder's Smart Folders while the nev OS was just released ?). Finally, all of this would not be possible without talentuous programming, but also real humanity and love of the Mac behind. James Thomson is one of the finest, and also most generous (and funniest) developers I've dealt with. So far, my most recommended piece of software o the Mac if you want to amaze people while enhance your productivity with a really *usable* Dock. You need it. Period.*

3. More coming soon... Hopefully!

TLASYSTEMS

[Home | About | What's New? | History | Download | Register | Support]

Last modified 15th May 2005

ABOUT
WHAT'S NEW
HISTORY
DOWNLOAD
REGISTER
SUPPORT
FORUMS

A Short History of DragThing - Ten Years Of Tidying Up Your Desktop

May 2005 marks the tenth anniversary of the original release of DragThing 1.0. Ten years is a long time to be working on one piece of software, so I thought it was only appropriate to celebrate the occasion!

In the beginning...

This page will be updated over the next ten days with various bits of trivia and previously undisclosed secrets about DragThing that you won't find anywhere else.

If you are new to DragThing, you can find out more information about it here, and download a copy of the latest DragThing 5.6 with Tiger support here to try it out.

It's okay, we'll wait for you to come back before we start.

- 10 Things You Didn't Know About DragThing
- 10 Year Timeline
- 10 Killer Apple Technologies Of The Last 10 Years That DragThing Didn't Support
- 10 Years Of Screenshots
- 10 Applications Used To Make DragThing
- 10 Comments

There have been thirty-nine different releases of DragThing so far - if you've used DragThing since the very beginning or if you have any questions, I'd like to hear fro you!

Send me a comment to the address below, or post on the DragThing Forums and I'l add it to this page.

admin@dragthing.com

Here's to the next ten years!

10 Things You Didn't Know About DragThing

1. The official story of DragThing's name is that I was working on my drag and drop based dock during the summer of 1994, and a friend of mine named Dai asked me something along the lines of "have you finished that drag thing yet? and the name stuck. However, since I have told that story so many times, I can't remember if it is actually true or not, and I suspect it isn't.

2. The name was originally intended to be a parody of the many "InterCaps" product names of the time, especially in the Apple world - AppleScript, QuickTime, MacWrite and so on. I'd never originally intended to make any money with DragThing, so it didn't matter - or so I thought - that it had a marketable name. However, DragThing 1.0 proved very popular and I was stu with it. Over the years, quite a few products appeared from other developers with the word "Thing" at the end which amused me to no end.

3. DragThing was originally freeware, or more accurately "coolware". The idea w if you liked it and used it, you were supposed to send me something cool. Mos commonly this involved T-shirts, but I got a lot of interesting stuff sent to me from all around the world. Some time later, I got an email from a guy at a ver large advertising company who explained that his finance department didn't understand the concept of coolware, and would it be okay if they just sent a large cheque instead? And so, capitalism was invented. A beneficial side effect of this change was that it somewhat alleviated the mounting T-shirt storage crisis - to the great delight of my significant other.

4. Early versions of DragThing used to expire a year after they were released, ar started warning people to download a new copy a month prior to that. I did th because I was fed up with people running ancient versions and asking about bugs that I had fixed years ago. Some people thought it was a cunning schem that I had planned years in advance to make people to upgrade to the shareware version, but it wasn't. Honest!

5. A certain utility software company once offered to buy the exclusive rights to DragThing for $10,000. I respectfully declined, and they bought out one of my similarly-named competitors instead. When DragThing became shareware son time later, it soon became clear that this had been the correct decision!

6. DragThing Lite, the simplified version of DragThing which shipped with DragThing 2, was originally written to be included with Mac OS 8.5 instead of the AppSwitcher. The deal with Apple fell apart quite quickly, but the code survived.

7. To enable the hidden debug mode in DragThing 2, open the Preferences windo and type on the arrow keys: up up down down up up up up.

This was the same code that enabled the debug mode in the game Sonic The Hedgehog 3 on the Sega Genesis/Megadrive.

8. The reason there wasn't much DragThing development between 1998 and 20(is because I was then part of the team working in secret on the Finder and Ma OS X Dock at Apple. I felt then that the Dock would ultimately replace DragThing. However, for a variety of reasons, I left Apple shortly after Aqua a the Dock were announced in January 2000, and very little of my code survive: in the current Dock. I moved back to Scotland, and work started on DragThin soon after.

   Update: I'm reliably informed that *none* of my original code survives in the current Dock. Thanks John for rubbing that in :-)

9. DragThing 3.0 was an unreleased version of DragThing 2.9 that had been adapted to run on early developer previews of Mac OS X. Because of Aqua, I realised I would need to rewrite much of the user interface code, and so I started working on DragThing 4 instead.

10. DragThing 4 on Mac OS X contained the coolest easter egg in the world, but fewer than ten people have seen it.

    Update: Many people have asked me what it is. I would love to tell you, becau it was really cool. But there would be *repercussions.*

10 Year Timeline

Some notable events over the last ten years:

1994	Work starts on DragThing 1.0 in the summer. PowerPC based Macintoshes first introduced. System 7.5 released.
1995	DragThing 1.0 is first released to the public in May. Cyberdog and OpenDoc announced at WWDC. Windows 95 released.
1996	I start working for a well known computer company. Apple buys NeXT.
1997	DragThing 2.0, the first shareware version, is released in July.

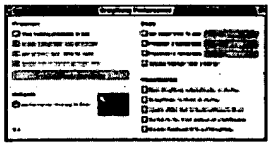
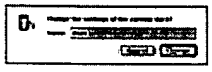
	Mac OS 8 released.
1998	The official dragthing.com site goes online in February. iMac introduced. Mac OS X announced.
1999	A quiet year for DragThing due to work on that *other* dock. iBooks and Airport introduced. Power Mac G4 ships.
2000	I stop working for a well known computer company. Aqua shown publicly for the first time.
2001	DragThing 4.0 for Mac OS X is released a few days before Mac OS X is. iPod revealed.
2002	DragThing 4.5 adds full support for Mac OS X 10.2 "Jaguar". LCD iMac ships with a bundled copy of PCalc.
2003	DragThing 5.0 adds full support for Mac OS X 10.3 "Panther". Safari introduced. iTunes Music Store opens its doors.
2004	DragThing 5.5 finally adds support for spring-loaded folders! 30" cinema display unveiled. iMac G5 ships.
2005	DragThing 5.6 adds full support for Mac OS X 10.4 "Tiger".

10 Killer Apple Technologies Of The Last 10 Years That DragThing Didn't Support

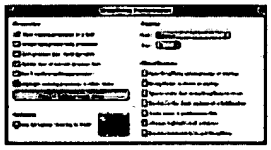
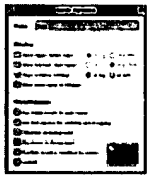
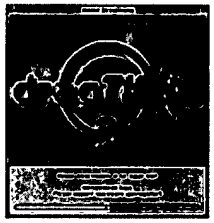
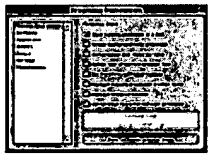
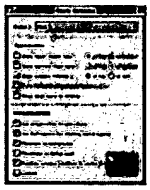
1. Copland
2. Rhapsody
3. OpenDoc
4. PowerTalk
5. QuickDraw GX
6. QuickDraw 3D
7. Publish & Subscribe
8. Newton OS
9. Dylan
10. Cocoa

10 Years Of Screenshots

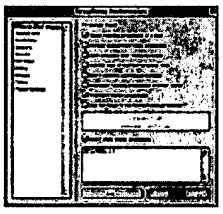
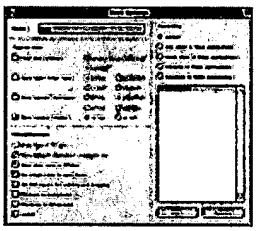
Click the images below to see how things have evolved:

	About	Preferences	Dock Options
1995			
1996			

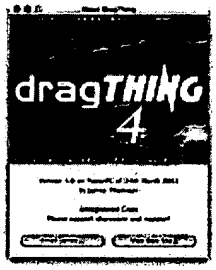
1997

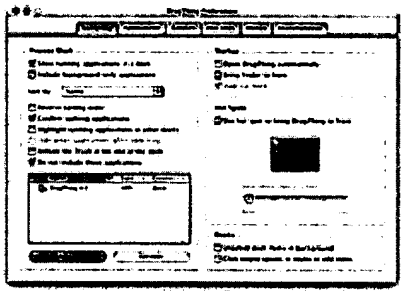
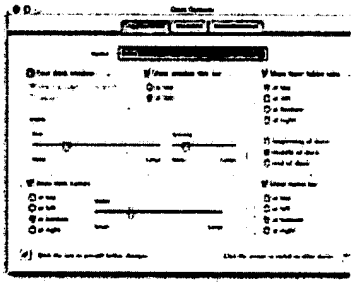






2000

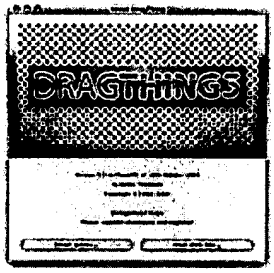



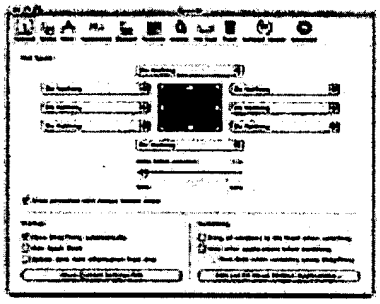
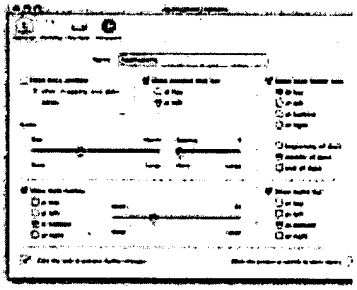
2001



2003



10 Applications Used To Make DragThing

1. Metrowerks Codewarrior - the development environment I've used for the last ten years
2. Apple Xcode - the development environment I'm likely to be using for the nex ten years, unfortunately
3. Bare Bones BBEdit - simply the best text editor
4. Adobe Photoshop - the best image editor, but certainly not simple
5. Resorcerer - at least it runs natively on Mac OS X!
6. ResEdit - the original. Still runs on Tiger
7. PowerGlot - so much nicer than AppleGlot
8. Iconographer - because everybody needs icons
9. Installer VISE / FileStorm - for installers past and disk images present
10. PCalc - the only software older than DragThing

10 Comments

1. From Cletus Waldman at ResExcellence:

   You've read this many times I am sure about various pieces of software. "It is the first item i install when I load a new system." But that is what is done on i machines with this little jewel. There is nothing I have ever found to replace DragThing. Nothing! Sure there are alternatives out there in third party space None with the feature set of DT. I happen to be a point and click person and r a keyboard shortcut guy. So this fits my work style perfectly and flawlessly.Th amazing thing about it however, is that it is such a dynamic and customizable program that one can literally have the best of both worlds— both point and click and keyboarding. It seems it is always literally one step ahead of the curve. I have been fortunate to be on the beta team also, so I know the care and work James puts in on this program. James you have one great little program there. I congratulate you, and wish you continued success in this wo of one shot shareware wonders.

2. From Guillaume Gete:

   *I began starting working with DragThing since 1995, or something like that. A to say that DT is one of the most stable pieces of software I've ever seen, eve at the time of Classic Mac OS, is an understatement. Funny to see that one of my oldest files on my Mac is still the DragThing preferences file from 1996... That could be a proof by itself. The fact is, from all the applications and utilitie I have tested for years on the Mac platform, DragThing is maybe the one with the most respect for the Mac itself, whether it is in its programming execution its carefulness of using the best Mac OS technologies in the best possible way. (do you know so many applications that on the 29th of April could use Quartz Extreme effects, AppleScript and support Finder's Smart Folders while the nev OS was just released ?). Finally, all of this would not be possible without talentuous programming, but also real humanity and love of the Mac behind. James Thomson is one of the finest, and also most generous (and funniest) developers I've dealt with. So far, my most recommended piece of software o the Mac if you want to amaze people while enhance your productivity with a really *usable* Dock. You need it. Period.*

3. More coming soon... Hopefully!

TLASYSTEMS

[Home | About | What's New? | History | Download | Register | Support]

Last modified 15th May 2005

Copland

From Wikipedia, the free encyclopedia.

This article is about the Copland project at Apple Computer. For the American composer see Aaron Copland.
For the movie starring Sylvester Stallone and Harvey Keitel see Cop Land.

Copland was a project at Apple Computer to create an updated version of the Macintosh operating system. Begun in earnest in 1994, it was abandoned in August of 1996.

Contents

Background

In 1989, managers at Apple had a meeting to plan the future course of Mac OS development. Ideas were written on index cards; features that seemed simple enough to implement in the short term (like adding color to the user interface), were written on blue cards, while more advanced ideas (like an object oriented file system) were written on pink cards. Development of the ideas contained on both sets of cards was to proceed in parallel. The projects were known simply as "blue" and "pink". Apple intended to have the blue team (which came to call itself the "Blue Meanies" after characters in Yellow Submarine) release an updated Mac OS in the 1990-91 timeframe, and the pink team to release an entirely new OS around 1993.

The blue team delivered what became known as "System 7" only slightly late near the end of 1991, but the pink team suffered from second-system effect and continued to slip its release into the indefinite future. Eventually Apple semi-abandoned the pink project by spinning it off to form Taligent. This left Mac OS in a bad position. Originally intended to support a single user running a single application on a non-networked machine with a floppy disk for storage, many parts of the operating system simply did not scale well to the increasing demands of users. In particular the architecture of QuickDraw made it very difficult to introduce multitasking into the system, and the file system was particularly inefficient when used with hard disk drives.

Several attempts were made by various teams to address these issues with an updated operating system, but they ran afoul of internal politics and turf wars. John Sculley, Apple's CEO during this period, largely ignored managing of the company itself while he concentrated on sales and marketing. As a result the engineering departments had no clear direction.

Design

With System 7.5 released in autumn 1994, Apple management decided that the decade-old Macintosh operating system had run its course, and an entirely new operating system with more advanced features would be needed for the platform to compete with upcoming releases of Microsoft Windows.

The result was a next-generation operating system, code named "Copland" after composer Aaron Copland. Copland was to run Mac OS on top of a microkernel named Nukernel, which offered modern versions of Mac services such as the file system and networking. Programs written for the new system would run as first-class citizens, able to call other programs using high-speed interapplication communications to provide services. One important program was the **blue box**, which essentially encapsulated an existing System 7 OS inside a single process in a single address space. Older Mac programs would run inside the blue box, as "cooperative tasks" that used non-re-entrant Toolbox calls. The microkernel would isolate access to memory and thereby increase stability; if a new application crashed it simply needed to be restarted, and at worst a "classic application" would require the blue box as a whole to restart. The days of applications taking down the whole system would be over.

The trick was to make all of this fit into the memory of existing Macs. Running a number of copies of System 7 would simply not work due to its large size. Copland attempted to address this by introducing a large number of shared libraries and a fiendishly complex memory management scheme. It was intended that Copland would only be perhaps 50% larger than the existing Mac OS (now well into the 2MB range), a reasonable goal given the ever-increasing amount of RAM being incorporated into the average machine.

Another key feature of Copland was that it would be completely PowerPC "native." System 7 had been recompiled on the PowerPC with great success, but the system still relied on the processor looking like a member of the Motorola 68000 family. In particular the interrupt handlers in the Mac OS had to be emulated, requiring an expensive call into the OS to translate these to the PowerPC's much simpler system. Removing this limitation would allow Copland-native applications to run much faster, as much as 50% in many cases, with no special effort on the part of the developers.

Development

Parts of Copland, most notably an early version of the new file system, were demonstrated at Apple's Worldwide Developers Conference in May 1994. Apple also promised that a beta release of Copland would be ready by the end of the year, for full release in early 1995, and that an even more advanced and "fully-modern" successor code-named Gershwin would follow in 1996. Throughout the year, Apple released a number of mockups to various magazines showing what the new system would look like, and commented continually that the company was fully committed to this project. By the end of the year, however, the developer release was nowhere in sight.

At WWDC '95 Apple's new CEO, Gil Amelio, talked exclusively about Copland, now known as **MacOS 8**, repeatedly stating that it was the only focus of Apple engineering. Amelio announced that it would ship to developers only a few months later at the end of the summer, with a full release planned for late fall. Very few demos of the system were actually shown at the conference. Instead what was demonstrated were various pieces of the technology and the user interface that would go into the system, such as a new file management dialog. Little of the technology of the core system was demonstrated; the new file system that had been shown a year earlier was absent here.

After a number of people at the show complained about the lack of sophistication of the microkernel, notably the lack of multithreading, Amelio came back on stage at the end of the show and announced that they would be adding that to the feature list. This implied that the system was nowhere near ready, as such a feature is so fundamental to a kernel that it would be impossible to add it so close to the shipping date.

In August 1995, "Developer Release 0" was sent to a small number of selected partners. The system was so unstable that it was impossible to develop on. Far from demonstrating improved stability, it often crashed after doing nothing at all. In October, Apple moved the target delivery date to "sometime," perhaps 1997. One of the groups most surprised by the announcement was Apple's own hardware team, who had been waiting for Copland to allow the PowerPC to truly shine. Also, none of the demo technologies were included with the developer

releases.

Cancellation

Later that summer the situation was no better, and Amelio realized something serious had to be done. He lured Ellen Hancock away from her management position at IBM to take over engineering and get Copland development back on track. After a few months on the job she realized the situation was hopeless; given current development and engineering, Copland would never ship at all. She suggested that development continue on the existing Mac OS to improve its stability, while looking outside the company for a new operating system.

In August 1996, just as "Developer Release 1" was being prepared, Apple officially cancelled Copland. Among the reasons given were the slow pace of development and the many technical problems remaining to be solved. Plans for Gershwin were also scrapped; development on Gershwin had not yet begun. Hancock noted that in discussions with developers it became clear there was no Copland/Gershwin plan at all, although everyone seemed to know that "Gershwin would be the fully modern Copland". In fact there was no one in the company that seemed to know exactly what that meant, and no engineering had been dedicated to producing it.

Following Hancock's plan, development of System 7.5 continued with a number of technologies originally slated for Copland being rolled into the base OS. These included *themes* and drag-and-drop text. Eventually they released a number of such upgrades as Mac OS 8, in order to deliver on their promise of having something called Mac OS 8 running on all existing machines and thereby avoiding a lawsuit.

At the same time, Hancock's other plan was also followed, and Apple started looking for a third-party product to buy. After lengthy discussions with Be, many were surprised when Apple bought NeXT Computer in December 1996. The project to port OpenStep to the Macintosh platform was named **Rhapsody** and was eventually released as Mac OS X, which also uses the "blue box" concept to run applications written for older versions of Mac OS.

A number of features seen in Copland demos, including its advanced Find command, built-in Internet browser, and support for video-conferencing, have reappeared in recent releases of Mac OS X.

See also

- Mac OS history

External Links

- Copland in Depth (http://mlagazine.com/modules.php?op=modload&name=News&file=article&sid=150&mode=thread&order=0&thold=0)

Retrieved from "http://en.wikipedia.org/wiki/Copland"

Categories: Failed Apple initiatives | Operating systems

OpenStep

From Wikipedia, the free encyclopedia.

OpenStep is an open object-oriented API specification for an object-oriented operating system that uses any modern operating system as its core, principally developed by NeXT. It is important to recognize that while **OpenStep** is an API specification, **OPENSTEP** (all capitalized) is a specific implementation of this OpenStep developed by NeXT. While originally built on a Mach-based Unix (such as the core of NeXTSTEP), versions of OPENSTEP were available for Solaris and Windows NT as well. Furthermore the OPENSTEP libraries (the libraries that shipped with the OPENSTEP operating system) are in fact a superset of the original OpenStep specification.

Contents

History

The OpenStep API was created as the result of a 1993 collaboration between NeXT Computer and Sun Microsystems, allowing this cut-down version of NeXT's NeXTSTEP operating system object layers to be run on Sun's Solaris operating system (more specifically, Solaris on SPARC-based hardware). Most of the OpenStep effort was to strip away those portions of NeXTSTEP that depended on Mach or NeXT-specific hardware being present. This resulted in a smaller system that consisted primarily of Display PostScript, the Objective-C runtime and compilers, and the majority of the NeXTSTEP Objective-C libraries. Not included was the basic operating system, or the display system.

The first draft of the API was published by NeXT in summer 1994. Later that year they released an OpenStep compliant version of their flagship operating system NeXTSTEP running on several of their supported platforms and rebranded it OPENSTEP. OPENSTEP remained NeXT's primary operating system product until they were purchased by Apple Computer in 1997. OPENSTEP was then combined with technologies from the existing Mac OS to produce Mac OS X.

Sun never seemed terribly interested in the product, likely a result of the NIH syndrome. In fact it's somewhat unclear why they were ever interested, although it appears it was an attempt to "get in" on the object-oriented operating system market before Microsoft released its plans for the object-oriented **Cairo** OS (which never happened). Nevertheless they started their port to Solaris some time in 1994, and released it in 1996. When Sun started work on Java just after this point, Solaris OpenStep was never seen again.

Description

The API OpenStep contrasts with the earlier NeXTSTEP primarily in five ways:

- OpenStep describes only the upper-level libraries and services (like Display PostScript), whereas NeXTSTEP referred to both these libraries and the operating system as well.
- removal of any code depending entirely on the Mach kernel, so that OpenStep could be run on top of any reasonably powerful operating system.
- a significant amount of effort was put into making the system "endian-free", an issue NeXT had already faced during a port of NeXTSTEP to the Intel platform.
- low-level objects such as strings were represented with C data types in NeXTSTEP, whereas in OpenStep a number of new classes (NSString, NSNumber, etc.) were introduced to support endian-conversion as well as provide added functionality. This had ripple-effects throughout the API, mostly for the better. Other algorithms to deal with low-level data and general fundamental functionality such as key programming algorithms (for example implementing classes to represent array structures (NSArray), and to deal with the processing of files (NSFileHandle)) were changed and made device independent. This set of classes (a *framework*) was called the *Foundation Kit*, or just Foundation for short.
- memory management did evolve from a simple alloc/free mechanism to a new retain/release paradigm: If a piece of code needs to keep an object valid it retains it, and when it doesn't need it anymore, it releases it.

The API specification itself is comprised of the two main sets of object oriented classes: the GUI and graphics front-end known as the **Application Kit**, and the aforementioned Foundation Kit.

However, OpenStep also specified the use of Display PostScript, a versatile and powerful PostScript-based method of drawing windows and graphics on screen. NeXT, with its devotion to implementing object-oriented solutions, thought the method of *pswraps*, interfacing C code to Display PostScript, acted in an encapsulative way and could be thought of its use as being somewhat object oriented like. The Application Kit, Foundation, and Display PostScript comprise the three key technologies in the OpenStep specification; however, Display PostScript was featured in older NeXT technologies, such as NeXTSTEP.

Building On OpenStep

The standardization on OpenStep also allowed for the creation of several new library packages that were delivered on the OPENSTEP platform. Unlike the operating system as a whole, these packages were designed to run stand-alone on practically any operating system. The idea was to use OpenStep code as a basis for network-wide applications running across different platforms, as opposed to using CORBA or some other system.

Primary among these packages was PDO (*Portable Distributed Objects*). PDO was essentially an even more "stripped down" version of OpenStep containing only the Foundation Kit technologies, combined with new libraries to provide remote invocation with very little code. Unlike OpenStep which defined an operating system that applications would run in, under PDO the libraries were compiled into the application itself, creating a stand-alone "native" application for a particular platform. PDO was small enough to be easily portable, and versions were released for all major server vendors.

PDO became somewhat infamous in the mid-90's when NeXT staff took to writing in solutions to various CORBA magazine articles in a few lines of code, whereas the original article would fill several pages. Even though using PDO required the installation of a considerable amount of supporting code (Objective-C and the libraries), PDO applications were nevertheless considerably smaller than similar CORBA solutions, typically about 1/2 to 1/3rd the size.

The similar **D'OLE** provided the same types of services, but presented the resulting objects as DCOM objects, with the goal of allowing programmers to create DCOM services running on high-powered platforms, called from Microsoft Windows applications. For instance one could develop a high-powered financial modelling application using D'OLE, and then call it directly from within Microsoft Excel. When D'OLE was first released, OLE by itself only communicated between applications running on a single machine. PDO enabled NeXT to demonstrate Excel talking to other Microsoft applications across a network before Microsoft themselves were able to

implement this functionality.

Another package developed on OpenStep was EOF (*Enterprise Objects Framework*), a tremendously powerful (for the time) object-relational mapping product. EOF became very popular in the enterprise market, notably in the financial sector where OPENSTEP caused something of a minor revolution.

OPENSTEP

NeXT's first operating system was NeXTSTEP, a sophisticated Mach-UNIX based operating system that was ported to run on several architectures (PA-RISC, SPARC, i386 and 68k). However, NeXT's new direction for NeXTSTEP was to free the operating system libraries from being tied to UNIX and becoming more device independent.

NeXT completed an implementation of OpenStep on their existing Mach-based OS and called it **OPENSTEP**. It was, for all intents, NeXTSTEP 4.0, and still retained flagship NeXTSTEP technologies (such as DPS, UNIX underpinnings, user interface characteristics like the Dock and Shelf, and so on), and retained the classic NeXTSTEP user interface and styles. OPENSTEP was further improved, in comparison to NeXTSTEP 3.3, with vastly improved driver support – however the environment to actually write drivers was changed with the introduction of the object-oriented DriverKit.

OPENSTEP supported Intel x86, Sun's SPARC and NeXT's own Motorola based 68k architectures, while the HP PA-RISC version was dropped. Unlike the Solaris port, which ran on top of the platform's "native" OS, these versions continued to run on the underlying Mach-based OS used in NeXTSTEP. OPENSTEP became NeXT's primary OS from 1995 on, and was used mainly on the Intel platform. In addition to being a complete OpenStep implementation, the system was delivered with a complete set of NeXTSTEP libraries for backward compatibility. This was an easy thing to do in OpenStep due to library versioning, and OPENSTEP did not suffer in bloat because of it.

OPENSTEP Enterprise

NeXT also delivered an implementation running on top of Windows NT 4.0 called **OPENSTEP Enterprise** (often abbreviated OSE). This was an unintentional demonstration on the true nature of the portability of programs created under the OpenStep specification. Programs for OPENSTEP/Mach could be ported to OSE with little difficulty. This allowed their existing customer base to continue using their tools and applications, but running them inside the Windows system which many of them were in the process of switching to. Never a clean match from the UI perspective -- probably due to OPENSTEP's routing of window graphics through the Display Postscript server, which was also ported to Windows -- OSE nevertheless managed to work fairly well and allowed OpenStep to exist for perhaps another year.

OPENSTEP and OSE had two revisions (and one major one that was never released) before NeXT was purchased by Apple in 1997.

Rhapsody

Rhapsody was a reworked OPENSTEP for the Mac, and formed the start of the process that resulted in Mac OS X. It is OPENSTEP with the "Classic" interface and could be regarded as OPENSTEP 5. Two versions of Rhapsody were released to developers. The first, Developer Preview 1, ran only on Intel hardware. The second revision, DP2, was the first version to include support for limited set of Mac hardware.

Mac OS X

After Apple purchased NeXT, OPENSTEP became the basis for their operating system, Mac OS X. Mac OS X's programming environment is essentially OpenStep (with certain additions such as XML property lists and URL classes for Internet connections) with Mac OS X ports of the development libraries and tools, now called Cocoa. Mac OS X could be considered OPENSTEP 5 or 6, and has become the single most-used Unix in the world.

GNUstep

Main article: GNUstep

GNUstep was suggested at the time of NeXTSTEP (predating OPENSTEP).

With OPENSTEP and OSE being purchased by Apple, who effectively ended the commercial development of implementing OpenStep for other platforms, GNUstep is an ongoing open source project aiming to create a free software implementation of the OPENSTEP libraries to Linux/Unix based systems.

The Foundation and AppKit libraries are completed with the exception of a few classes which are rarely used. GNUstep also features a fully functional development environment.

Retrieved from "http://en.wikipedia.org/wiki/OpenStep"

Categories: NeXT | BSD | Mach | Mac OS X | Application programming interfaces

OpenDoc

From Wikipedia, the free encyclopedia.

OpenDoc was a multi-platform software componentry framework standard for compound documents, inspired by the Xerox Star system and intended as an alternative to Microsoft's object linking and embedding (OLE).

Description

The basic idea of OpenDoc was to create small single-purpose reusable components responsible for a specific task, such as text editing, bitmap editing or browsing an FTP server. OpenDoc provided a framework in which these editors could run together, and provided a document format for storing the data being generated by each component. These documents could then be opened on other machines, where the OpenDoc frameworks would substitute suitable editors for each part, even if they were from different vendors.

In this way users could "build up" their documents from parts. Since there was no main application and the only visible interface was the document itself, the system was known as *document centered.* It was envisioned that OpenDoc would allow smaller 3rd party developers to re-enter the office software market, able to build one good editor instead of having to provide a complete suite.

OpenDoc was initially created by Apple Computer in 1992 after Microsoft approached Apple asking for input on a proposed OLE II project. Apple had been experimenting with software components internally for some time, based on the initial work done on its publish and subscribe linking model and the AppleScript scripting language, which in turn was based on the HyperCard programming environment. Apple reviewed the Microsoft prototype and document and returned a list of problems they saw with the design. Microsoft and Apple, who were highly competitive at the time, were unable to agree on common goals and did not work together.

At about the same time a group of 3rd party developers had met at WWDC '91 and tried to hammer out a standardized document format, based conceptually on the Amiga's IFF. Apple became interested in this work, and soon dedicated some engineers to the task of building, or at least documenting, such a system. Initial work was published on the WWDC CDs, as well as a number of follow-up versions on later developer CDs. A component document system would only work with a known document format that all the components could use, so it was only a matter of time before the standardized document format was pulled into the component software effort. From then it quickly changed from a simple format using tags to a very complex object oriented persistance layer called Bento.

Initially the effort was code named "Exemplar", then "Jedi" and "Amber" before being released under the name OpenDoc. The development team realized in mid-1992 that an industry coalition was needed to promote the system, and created the *Component Integration Laboratories* ("CI Labs") with IBM and WordPerfect. In 1996 the project was adopted by the Object Management Group.

Kurt Piersol from Apple Computer was the best known of the architects of OpenDoc, but Jed Harris (later president of CILabs) was just as critical to the early designs. Mark Ericson from WordPerfect provided the vision for a port to Windows that included seamless interoperability between OpenDoc and OLE.

OpenDoc was one of Apple's earliest experiments with open source development methods. Apple and its partners did not release the source code for wide reuse, but did make the complete source available to developers for feedback and for testing and debugging purposes.

Release

OpenDoc was initially released to run on Mac OS System 7.5 to provide a document-based, rather than application-based, computing experience. Documents were made of modular **parts**, which could contain different types of content, such as pictures, spreadsheet information, text or even Quicktime multimedia elements. Parts relied on specific **part editors** to allow the user to modify the content, or **part viewers** to display the content without allowing the user to edit the part.

OpenDoc's primary distinction from other compound document architectures lay in the depth of its support for dynamic media. OpenDoc containers could include embedded live content, and could perform arbitrary real-time composition of the content. The architecture used a design pattern which insulated container from embedded content using intermediate objects, greatly enhancing interoperability and simplifying testing of part handlers. Any part could serve as a **container** for any other part.

The WAV word processor was a semi-successful OpenDoc word processor; the Numbers & Charts package was a spreadsheet and 3D real-time charting solution from Adrenaline Software, the CyberDog web browser was created by Apple as an OpenDoc application; the Nisus Writer software by Nisus incorporated OpenDoc. RagTime, a completely integrated office package with spreadsheet, publishing and image editing was ported to OpenDoc shortly before OpenDoc was cancelled. Apple's 1996 release of ClarisWorks 5.0 (the predecessor of AppleWorks) featured support for OpenDoc components.

From IBM's involvement in Taligent, there was an implementation of OpenDoc in OS/2 Warp 4. IBM also contributed a large amount of development to the underlying object technology, the Common Object Request Broker Architecture (CORBA).

OpenDoc had several hundred developers signed up, but the timing was poor. Apple was losing money, the Java programming language, JavaBeans and Web-based applications were all being hyped as the next new way of building applications. Before long, OpenDoc was scrapped, with Steve Jobs noting that they "put a bullet through [OpenDoc's] head", and the entire team was laid off in a big reduction in force in March 1997. Other sources say that Microsoft hired away the key developers and no one was left to continue development. [1] (http://www.swiss.ai.mit.edu/~bob/clarisworks.php#oregon)

External links

- Apple's OpenDoc Documentation (http://developer.apple.com/documentation/macos8/Legacy/OpenDoc/opendoc.html)
- C2 about OpenDoc (http://c2.com/cgi/wiki?OpenDoc)
- ClarisWorks 5.0 with OpenDoc components (http://www.swiss.ai.mit.edu/~bob/clarisworks.php#oregon)
- Overview of OpenDoc (http://www.sundialsystems.com/articles/opendoc.html)

Retrieved from "http://en.wikipedia.org/wiki/OpenDoc"

Categories: Apple software | Failed Apple initiatives | IBM software

Object linking and embedding

From Wikipedia, the free encyclopedia.

Object Linking and Embedding (OLE) is a Microsoft technology.

A distributed object system and protocol from Microsoft, also used on the Acorn Archimedes. OLE allows an editor to "farm out" part of a document to another editor and then reimport it. For example, a desktop publishing system might send some text to a word processor or a picture to a bitmap editor using OLE. The main benefit of using OLE, next to reduced file size is the ability to create a master file. References to data in this file can be made and the master file can then have changed data which will then take effect in the referenced document.

It was initially used primarily for copying and pasting data between different applications, especially using drag and drop, as well as for managing compound documents. It later evolved to become an architecture for software components known as the component object model (COM), and later DCOM.

OLE, Version 1.0 was released in 1990.

> *This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.*

Retrieved from "http://en.wikipedia.org/wiki/Object_linking_and_embedding"

Categories: Software stubs | FOLDOC sourced articles

Apple Open Collaboration Environment

From Wikipedia, the free encyclopedia.
(Redirected from PowerTalk)

Apple Open Collaboration Environment, or **AOCE** (sometimes **OCE**), was a collection of messaging-related technologies introduced for the Mac OS in the early 1990s. It included the **PowerTalk** mail engine, which was the primary client-side interface to the system; the **PowerShare** mail server for workgroup installations; and a number of additional technologies such as **Open Directory**, encryption and digital signature support.

AOCE/PowerTalk was heavily marketed between 1993 and 1995, but the hardware requirements meant that most users couldn't even install it, let alone use it. Developers were likewise stymied by the complex system, and by users who didn't seem to expect to have to pay for something that was otherwise intended to be "built in". In 1996 Apple Computer quietly dropped their efforts to market AOCE, and the project quickly disappeared.

History

Development of AOCE started in 1989, largely the "pet project" of Gursharan Sidhu, engineering lead at Apple for LaserWriter, AppleShare and related networking products. At the time, during John Sculley's "don't bother me with technology" era, practically any mid-level manager could arrange to have projects funded if they had enough "pull". This was certainly the case for Sidhu, who was well respected within Apple.

The project started by taking a "20,000 foot overview" of existing mail systems, and trying to find common concepts and problems. The key conclusion they came to was that e-mail systems were confused about their own purpose; their key task is to provide a mechanism for store-and-forward delivery of *things* to *places*, but existing systems invariably delivered *e-mail* to *people*. Compare this with the real-world postal service, which will deliver not only mail, but magazines, packages, large parcels, and even (in one example) building materials to a worksite.

They also found that existing e-mail systems shared a number of common problems. They tended to support plain text mail only, and rarely included any support for non-English characters. Support for mobile users was spotty at best, often relying on 3rd party "hacks" that were of dubious reliability. And they were all, without exception, based on a dedicated e-mail server that was typically complex to set up, and often "overkill" for small installations with only a few people in an office.

And finally none of the existing products could give the user what they really wanted: a single universal mailbox and a single universal address book. At the time savvy users would often have mailboxes on their corporate network, online services such as CompuServe or AppleLink, and perhaps a number of BBS systems as well. Each e-mail system used its own standards for collecting and storing information, forcing users to run multiple clients to access the different services. Although a single-mailbox system could be constructed by administrators with the use of e-mail gateways, these tended to be very expensive and technically challenging to maintain.

AOCE aimed to fix all of these issues at the same time. At "one end" of the system, AOCE focused on the underlying delivery and addressing systems, generalizing the e-mail concept so the system could be used to deliver anything from e-mail to word processor documents to print jobs. Addressing was another issue the market was struggling with, so AOCE would offer a single universal addressing mechanism and address book, one that could support not only people's e-mail addresses, but the addresses of things like printers and fax machines as well.

Remote users would be supported by native AppleTalk Remote Access, Apple's "standard" solution for supporting the AppleTalk protocol over modems. The system understood that users were not always connected

and delivery didn't have to happen until both the sender and recipient were online. Even on a LAN this would be valuable, as many people turn off their computers at night and the mail would have to wait until the next morning for delivery. For security over the potentially "open" phone lines, all communications could be secured using RSA encryption and digital signing, even on the local network.

AOCE would normally run peer-to-peer, with user's mail being stored locally on their machine. This allowed them to go mobile, as well as allowing small networks to be set up for zero cost. Users would be able to send documents directly to other users simply by dropping the address on the document, or vice-versa, bypassing a "message" at all -- the document would arrive, by itself, in the other user's mailbox. Delivery of e-mail, or anything else for that matter, would be handled entirely by plug-ins, allowing the user to collect mail from all of their sources and collect it into their single inbox. An optional server could be installed for performance and maintenance needs for those sites that required it.

In other words, AOCE was suffering from the second-system effect, where engineers spend considerable time designing a system that does *everything*. Invariably these projects fail as the demands are not only incredibly difficult to meet, but often fail to meet real user needs. Often the ideas themselves are good, but buried inside unusable implementations. Users would not learn how unusable until it was released.

By early 1993 the "client side" of AOCE was nearing completion. Apple started a pre-release marketing campaign, telling their larger customers and even 3rd party e-mail vendors that AOCE would soon arrive and change the market completely. The claim was that nothing else would be able to compete with its ease-of-use, power, and lack of maintenance overhead -- all hallmarks of "the Apple way".

AOCE was publically released in September 1993, part of the **System 7 Pro** bundle that also included AppleScript utilities.

In-use

When the product finally shipped after years of hype, users were dismayed to find that there was basically no way they could install it. An install of the various client-side components required a machine with 2.5MB of RAM minimum, and really needed 4MB to run well. This was the maximum available RAM in many Mac systems of the era. Removing unneeded components did little to address this, and AOCE and the other Apple technology du jour, QuickDraw GX, typically could not be run together because of a lack of memory. While newer machines were able to run AOCE more comfortably, as an e-mail system intended to be run on diverse networks of non-homogeneous machines the requirements greatly impaired market acceptance. Simply downloading and installing separate stand-alone client applications for each mail system the user actually had would use considerably less disk space, and had no constant memory footprint.

It was equally clear that while PowerTalk was generally an interesting system, a combination of design features made it frustrating to use in the real world. For instance, the addressing system was so deeply embedded into the core of the system that simply typing in a new address was an ordeal. First the user had to click on a button, select the address type, type it in, and then finally click OK to have it appear in the message. Disk usage was also ridiculous; each message was stored as a separate file, requiring 1k or more of space in an era where 40MB and 80MB disks were still common. The result was that only a few hundred letters would be enough to fill the free space on the drive.

Another annoyance was due to the fact that the Mac OS did not require users to log in, and thus the system could not know who a user was. This meant that documents had to be delivered to a user's *machine*. This, obviously, did not work well in the case where the user had two or more machines, making the concept of a universal mailbox annoying in practice. Backing up e-mail was likewise almost impossible; the mail was spread out over the network, some of it remote and unaccessable.

Even the remote access was doomed by feature interaction. To ensure that all messages were delivered in a reasonable time on a network where machines might appear and disappear at random (when they are turned on and off), AOCE had a 15-minute timeout in which it tried to deliver pending messages. However, if the user in question was using a dialup connection on a modem, this meant that delivering mail took of 15 minutes on the phone [grammar check??] if any of the recipients were not online.

Many of these problems were intended to be solved with the PowerShare server, which acted as an always-on, always responsive "super-peer". The basic AOCE protocol would notice these machines when attempting delivery, and send to them first, thereby eliminating the delays and centralizing storage and maintenance. Sadly the server was not ready in time for the release, and didn't ship for another year. When it did it was likewise slow and resource hungry, largely a side effect of various features of the Mac OS that made it unsuitable for server applications (not that it was designed for this role).

AOCE had one year in the sun at Apple's Worldwide Developers Conference in 1995 when it was finally gelling, but by this point almost everyone realized that the market was clearly moving towards SMTP-based internet mail as an almost universal format. By 1996 Apple had given up on AOCE, and started talking about the CyberDog project based on the OpenDoc platform. AOCE quietly disappeared.

Given reasonable staffing estimates over that time, it is possible that the project cost anywhere from $50 to $100 million at this point, thereby adding $50-100 to the price of every Mac sold.

Description

AOCE's **Open Directory** and related software introduced the concept of directory entries (such as business cards) as first-class desktop objects. This was used to create a drag-and-drop metaphor for mail, fax, and other directory-based activities. Each endpoint, a mail server for instance, was driven by a plug-in extension that was driven by a common AOCE-supplied queue and queue viewer. PowerTalk provided a set of standard forms for interacting with the items in the queues, and a common interface for mail, and a universal mailbox. Encryption was supported by a single "keychain" that remembered all your passwords and digital signatures, encrypting them together so only a single password needed to be remembered.

The system was designed in an era when there were many e-mail formats and services, including online services such as CompuServe and AppleLink, networking standards like X.400 and SMTP (internet mail) and LAN-based servers such as Microsoft Mail and QuickMail. In order to support this diverse environment, AOCE included a robust layered protocol stack that, in theory, could be used with practically any store-and-forward type of environment. This was used within AOCE not only to service mail, but faxes, printing and even directly sending files from one machine to another without enclosing them in a mail message or needing a file server.

Adoption of AOCE among 3rd party developers was slow due to a ferociously complicated API. The book documenting the system was larger than all of the books describing the rest of the pre-System 7 Macintosh put together. Adding a simple feature like "mail this document" to an application required wading through hundreds of pages of documentation, and writing a core AOCE component was many times more complex.

Several parts of the AOCE engine were useful on their own, notably the Keychain. However in order to get the keychain, you had to install all of AOCE, a cost the users were not willing to pay. Many years later the keychain was finally shipped as a stand-alone component in Mac OS 9. The encryption/signing support is arguably also useful (although not widely used in the "real world"), and it has reappeared in Apple's bundled Mail application starting with Mac OS X 10.2 Jaguar in August 2002.

Other concepts in AOCE remain intriguing and have yet to appear in other systems. For instance the idea of being

able to drag a document onto a user's address (which today would be standardized using a vCard) to directly pass the document on the next time that user appeared on the network seems like an excellent idea. Likewise the concept of peer-to-peer mail is something almost no systems offer, although the need for this in an era of ubiquitous mail servers is arguable.

Retrieved from "http://en.wikipedia.org/wiki/Apple_Open_Collaboration_Environment"

Categories: Failed Apple initiatives

QuickDraw GX

From Wikipedia, the free encyclopedia.
(Redirected from Quickdraw GX)

QuickDraw GX was a replacement for the QuickDraw graphics engine and Printing Manager, initially released in about January 1995. Version 1.1.1 was bundled with Macintosh System 7.5 later that year. Besides including a complete replacement for the traditional Macintosh printing architecture, it also introduced a rehousing of Type 1 fonts moving the PostScript outlines into the TrueType font file format. While the intentions behind these reorganizations were good--to make applications faster to make for the developer and more interoperable to use with system level text management and color management--there were too many existing applications that had developed their own architectures. Thus, the installation of GX introduced a host of incompatibilities that only succeeded in annoying users. This, coupled with opposition from the developer market and a lack of communication from Apple about the benefits of QuickDraw GX and why users should adopt it, led to the technology being sidelined. Mac OS 8 dropped support for the GX printing architecture, though the text management and color management architectures survived. Out of the text management architecture came the TrueType Specification and out of the color management architecture came the ICC International Color Consortium specification. With the advent of OS X, GX lives on in ATSUI Apple Type Services for Unicode Imaging, in ColorSync whose file format is identical to the file format of the ICC International Color Consortium, and in transparency which is visible in the user interface.

Contents

Graphics

Unlike QuickDraw, QuickDraw GX allowed for fractional coordinates. However, these were fixed-point values, rather than floating-point. This might have been because at the time GX was being developed (late 1980s to early 1990s), there was still a significant performance penalty in using floating-point arithmetic.

In addition to coordinates was the concept of the **gxMapping**. This was a 3-by-3 matrix which could express arbitrary linear transformations in two dimensions, including perspective distortions.

In the world of PostScript, features not available in the PostScript graphics model could be programmed by the developer. The programmability of PostScript led to unreliable printing performance, and was abandoned by Adobe in the PDF Portable Document Format language. The GX graphics architecture was built around a number of types of objects which were premade, though a full set of API calls was available for examining and manipulating them:

- a **gxShape** defined the basic geometry of a shape (for example, coordinates of control points for a curve, or the text content of a text object).
- a **gxStyle** defined elaborations of the basic shape geometry, such as line thickness, cap and join styles, fill pattern and text font.
- a **gxInk** specified how pixel values were to be computed when rendering the shape: besides specifying a

basic colour for the shape, this also included an elaborate *transfer mode* structure that could define a wide variety of functions of the initial and final destination pixel value.
- a **gxFont** represented a font, either one installed for systemwide use, or one installed on-the-fly by the current application for its own use. API calls allowed the interrogation of the properties of a font, including the determination of what encodings (Unicode, language-specific etc) it might support.
- a **gxProfile** was a representation of a ColorSync colour profile, used as part of the specification of a colour for drawing. GX integrated full support for colour matching at all stages of the drawing process, as well as support for non-RGB colour specifications (such as HSV, YUV and CIE XYZ).
- a **gxTransform** determined the relationship between the shape and the display device. Besides the clipping path and the gxMapping that transformed the shape before displaying on the output device, this object also specified hit-testing information that controlled responses to user clicks within the area of the shape.
- a **gxViewDevice** represented a block of pixel memory into which drawing would be rendered. This could be an actual on-screen display, or an offscreen block of memory. GX supported all the QuickDraw pixel layouts; this allowed both a GX view device and a QuickDraw GrafPort to point to the same pixels, thereby allowing applications to mix both sets of drawing calls.
- a **gxViewPort** was a logical destination for drawing. A gxTransform could specify a list of more than one of these; the shape would be drawn into all of them in a single `GXDrawShape` call.
- a **gxViewGroup** represented the connection between view devices and view ports. Each view port had a gxMapping specifying its relationship to the global coordinate system of the view group; and each view device had a gxMapping that specified its location and the size of its pixels with regard to view group coordinates. THere was a single prdefined view group which contained all on-screen view devices (and whose view ports effectively corresponded to on-screen windows); applications were free to create their own view groups for off-screen view devices and view ports.
- a **gxTag** allowed the attachment of arbitrary application-defined information to most of the above object types. Each tag had an OSType type code, but there could be multiple tags of the same type attached to the same object.

Shape types

GX shapes could be of various types:

- a straight line defined by its end points.
- a rectangle defined by its left, right, upper and lower bounds.
- a polygon defined by a sequence of vertex coordinates.
- a *curve* shape was a single quadratic Bézier curve defined by three control points.
- a *path* shape which was a sequence of quadratic Bézier curves. Each control point had an associated flag indicating whether it was "on-curve" or "off-curve". An on-curve point was a Bézier endpoint, while an off-curve point was a Bézier midpoint. If two successive off-curve points were encountered, then an implicit on-curve point was assumed to lie halfway between them. Two successive on-curve points defined a straight-line segment.
- a *bitmap* shape contained raster data in any of the supported pixel formats.
- a *picture* shape was a grouping of other shapes (possibly including recursive picture shapes), with the option of specifying additional transformations applying to the whole group.
- the various types of *typographic* shapes are described in the GX Typography section below.
- additional types which were perhaps not directly useful for drawing, but could be combined with other shapes in geometry calculations: the *empty* shape (drawing of which did nothing); the *point* shape consisting of a single point; and the *full* shape (of infinite extent).

Typography

The typography features of GX were integrated in the form of 3 types of gxShape:

- *Text shapes* were the simplest: these contained a single run of text rendered in a single font style.

- *Glyph shapes* were a way to use character shapes ("glyphs") as pure geometry, for instance as clipping paths.
- *Layout shapes* were the most elaborate. These could be divided into multiple runs with different font styles, even different language encodings and text directions. Thus, it was possible to embed a sequence of Arabic text, rendered right-to-left, within an outer sequence of left-to-right Roman text. Layout shapes unleashed the full power of contextual substitutions, kerning, variations and all the other capabilities of TrueType GX fonts. Their main restriction was that they were confined to a single line of text.

The GX API also provided hit-testing functions, so that for example if the user clicked on a layout shape in the middle of a ligature, or in the region between a change of text direction, GX itself would provide the smarts to determine which character position in the original text corresponded to the click.

TrueType GX

An important distinction in GX was drawn between a **character** and a **glyph**, a distinction also found in the Unicode Standard. A *character* was an abstract symbol from the character set of a writing system, such as the letter "f" in the writing systems of the Latin script. Whereas a *glyph* was a specific graphic shape from a particular font, whether the shape represented a single character or a set of characters. Thus, for example, the Hoefler Text font had glyphs to represent the letters "f" and "l". It also had another glyph to represent the *ligature* "fl", which could be automatically composed (instead of the individual glyphs) wherever the two abstract characters "f" and "l" occurred in sequence in the source text.

This distinction was important in that such contextual substitutions occurred at rendering time, without any changes to the source character string. Thus they had no impact on editing or searching of the text. PostScript Type 1 font files have one to one mapping only, and as ligatures are many to one mappings, they can not be inserted into the composition without changing the source character string, for instance, the ligature ffi is placed at the position of capital Y in Adobe font products, and "Adobe Offices" is composed by typing "Adobe O" <change font> "Y" <change font> "ces". In the layout the character string is broken, and in PDF made from streamed PostScript the characters f+f+i can only be reconstructed, if the name of the glyph follows a glyph naming list.

Contextual substitutions cam be controlled by enabling or disabling the composition options of a TrueType GX font in WorldText on the Mac OS 9 CD or in TextEdit in Mac OS X. Fonts commonly have features called "common ligatures" (such as the "fl" example), "rare ligatures" (such as inscriptional ME and MD ligatures", "archaic non-terminal s" (for automatically substituting the letter "s" with the archaic form that looked more like an "f", except at the ends of words), and even choices between entirely separate sets of glyph designs, such as more and less ornate forms.

The rules for performing contextual substitutions are implemented as state machines built into the font, and interpreted by the LLM Line Layout Manager, the counterpart of the CMM Color Management Module for ColorSync services. Text management in the operating system allowed QuickDraw GX to accept character strings with any mix of writing systems and scripts, and compose the strings automatically, whether the encoding was Unicode 1.0 or 8 bit and 8/16 bit encodings.

Another interesting feature was font *variations*, which were the GX equivalent of Adobe's "multiple master" fonts. Only, Adobe's idea was that you had to explicitly create an *instance* of the font (by specifying values for the variation axes) before you could use it. Whereas GX allowed you to specify the font directly for a layout style, and then dynamically vary the axis values and immediately observe the effect on the layout of the text.

Printing

Categories: Section stubs | Graphics software | Failed Apple initiatives | Apple software

QuickDraw 3D

From Wikipedia, the free encyclopedia.

QuickDraw 3D, or **QD3D** for short, is a 3D graphics API developed by Apple Computer, originally for their Macintosh computers, but delivered as a cross-platform system. QD3D provided a high-level API with a rich set of 3D primitives that was generally much more full-featured and easier to develop than low-level API's such as OpenGL or Direct3D. Below this was a cleanly-separated hardware abstraction layer known as **RAVE** that allowed the system to be ported to new hardware easily. On the downside, Q3D used a number of Apple-only ideas about how 3D hardware should work, and initially performed poorly due to the lack of hardware acceleration. Apple abandoned work on QD3D after Steve Jobs took over in 1998, and announced that future 3D support would be based on OpenGL.

Contents

3D in general

Most 3D toolkits concentrate on the lowest levels of the 3D rendering pipeline. These include describing the basic geometry of the "world", and systems for describing objects based on that geometry. Key to high performance in 3D applications is limiting the data that needs to be manipulated, and storing that in an efficient way in order to reduce the amount of memory that needs to be transferred to the display hardware during drawing. Computer bus operations can be many times slower than the processing of that data, so every effort must be made to reduce the overall load.

For instance, the OpenGL system consists primarily of a system for describing geometry in various ways that eliminate common vertexes from the data, along with commands for manipulating the objects and drawing them in the world. To further reduce traffic, OpenGL is a "stateful" system, in which the result of drawing an object into the world will depend on the state of the system set up in previous commands. The advantage to this approach is that most graphics share considerable amounts of state, so the setup information can be sent once, followed by many drawing operations. This also means that complex programs have to carefully track and restore state during drawing, an expensive operation that can actually lead to decreased performance if done too often.

Higher-level concepts of the 3D world are generally not included in low-level libraries. For instance, OpenGL has no system for storing the relationships between parts or building objects out of other objects. This is particularly important in 3D graphics, where objects are generally constructed out of a number of parts that are moved together to produce a larger model. This grouping, known as a scene graph, is typically left to the end-user to develop on their own, although a number of attempts to provide a standardized stateless scene graph were made several times over OpenGL's history. Even common issues such as interacting with the model or placing the view in a window are generally left to add-on libraries, in order to give the developer complete control over the system. However this also means that developers are writing the same basic code over and over again, just to get started.

To add to the problems, OpenGL was not *truly* low-level. It included about 250 calls in the basic API, many of which were used only in high-end rendering and offered little utility. The inclusion of these calls meant that the

API was too large to implement fully on consumer-level graphics cards of the era. This made the overall development task more difficult; it supported none of the high-level features that eased development, nor was it small enough to fully run on the hardware. In order to address the later issue, smaller subsets of OpenGL were developed, notably the Glide API and MiniGL.

QD3D

QD3D had the disadvantage of running on machines with considerably less power than the SGI boxes OpenGL was designed for. This guided the developers to cleanly separate the system into a high-level and low-level set of APIs, so that the lower levels could be implemented entirely in hardware. Since the high-level API was not intended to run directly on hardware, it could be made as feature packed as it wanted.

The result was that QD3D offered a considerably richer programming system for developers, one that dramatically reduced the workload for basic applications. QD3D provided functionality not only for storing and manipulating a scene graph, but also added a number of easy to use 3D objects, UI elements, and even input controllers. The suite allowed developers to add 3D support to their applications with ease, in a way that had a consistent UI when compared to other 3D applications using the platform. Apple additionally introduced the 3DMF file format in order to allow developers to exchange 3D models between applications, providing all the needed code for loading and saving 3DMF files.

Additional functionality included a "plug-in" rendering system, which allowed an application to render a scene in a variety of styles. Without changing the model or their code, developers could render the same scene interactively or (with suitable plug-ins) using methods such as ray-tracing or hidden-line.

The QD3D API was an "object like" system based on pure-C code. The various structures were carefully constructed to contain pointers to other important objects. Objects knew all the drawing state they needed, thereby eliminating a considerable amount of code that would normally be needed when developing under OpenGL. For obvious reasons the APIs were also tightly integrated with the Mac OS, allowing the developer to bring up an interactive 3D view in a window with little code. The developer's task was essentially to provide the models and "behavior" for the world, as opposed to the basic 3D framework that was needed for lower level libraries.

To contrast the two approaches, consider this sort of task that QD3D could directly support:

```
load the model "human",
load the style "t-shirt",
apply "t-shirt" to "human",
draw
```

Under OpenGL this task would resemble something more like:

```
load the contents of this file into a list of vertexes,
place the vertexes into a structure,
create a display list with the structure,
load this bitmap into the list,
load this bumpmap into the list,
etc....
draw
```

On the downside, QD3D's layering introduced performance issues. For instance, the system stored and automatically set state for every object before drawing. This made development much easier, but also made the performance drop in a way the developer had no direct control over. Under OpenGL the user was forced to handle

state themselves, but in doing so could often avoid the majority of state-setting commands and thereby improve performance.

Another area of concern is that the scene graph was hidden from view, and considerable improvements in rendering performance can be made by carefully "culling" the graph to remove those objects that are not in view. Although later releases of QD3D gained the ability to automatically perform visibility culling (based on the grouping of objects in the scene graph), OpenGL's lack of support for this feature typically forced developers to implement it from the start.

RAVE

In order to deal with the performance-critical rasterization operation, Apple designed QD3D to sit on top of a separate package known as **RAVE**. The result was something much closer to OpenGL -- more primitive to program, but with increased control over rendering.

Although RAVE did what it intended to do, the effort was essentially doomed. Good low-level 3D performance relies not only on the programmer to provide efficient models, but high-quality drivers for the hardware as well. This is where RAVE failed. Developers had repeated watched the "next big thing" come out of Apple only to see it be killed off in the next company reorganization, and were increasingly gun-shy of putting any effort into supporting Apple's latest developments. By 1996 the market generally felt Apple was doomed to bankruptcy, making matters considerably worse. Microsoft was at the same time trying to introduce their own similar library, Direct3D (D3D), and even though QD3D beat it to market and was technically superior, it was clear to everyone that it was basically being ignored in favor of D3D.

At the same time the expected market for 3D desktop applications simply never materialized. Most users stayed with their workstation-based modelers for performance reasons, and most other users have limited 3D needs. One of the few common uses was charting software, and even this is not at all universal. 3D games, on the other hand, took off at about this time, and drove the widespread availability of consumer-level 3D hardware.

Although RAVE was designed to be cross-platform, only Mac hardware developers (ATI and nVidia) produced drivers for it. This left any comparison between QD3D and alternative APIs one-sided, as outside of the Mac QD3D was forced to fall back to a software RAVE implementation.

Switch to OpenGL

As OpenGL gained traction on Windows (often credited to Id Software, who championed the API over D3D), hardware developers were increasingly designing future hardware against the future feature set planned for MS's D3D. Through its extension mechanism OpenGL was able to track these changes relatively easily, while RAVE's feature set remained relatively fixed.

At the Macworld Expo in January 1999, Apple announced that neither QuickDraw 3D nor RAVE would be included in Mac OS X. The company laid off the development staff in June of 1999, replacing the in-house technology with OpenGL after buying a Mac implementation and key staff from "Conix".

Today there remains no standard high-level API for 3D graphics. Several attempts have been made, including OpenGL++ and the SGI/Microsoft Fahrenheit graphics API, but none of these have made it to production.

After Apple withdrew support for QD3D, an Open Source implementation of the QD3D API was developed externally. Known as "Quesa", this implementation combines QD3D's higher level *concepts* with an OpenGL renderer. As well as cross-platform hardware acceleration, this library also allows the use of the QD3D API on

platforms never supported by Apple (such as Linux).

External links

Quesa (http://www.quesa.org/)
- Open Source QD3D implementation

QuickDraw 3D: A New Dimension for Macintosh Graphics
(http://www.mactech.com/articles/develop/issue_22/quickdraw.html)
Must-See 3-D Engines (http://www.byte.com/art/9606/sec11/art4.htm)
- compares QD3D, OpenGL and Direct3D

Retrieved from "http://en.wikipedia.org/wiki/QuickDraw_3D"

Categories: Graphics software | Apple software | Failed Apple initiatives | Application programming interfaces

Apple Newton

From Wikipedia, the free encyclopedia.

The **Newton** was an early personal digital assistant (PDA) developed by Apple Computer and sold from 1993 to 1999. It was based on the ARM processor, and featured handwriting recognition. Apple's official name for the device was *MessagePad*; the term *Newton* was Apple's name for the operating system it used, but popular usage of the word *Newton* has grown to include the device and its software together.

Contents

Overview

Newton was unsuccessful in the marketplace for two primary reasons: its high price (which went up to $1000 when models 2000 and 2100 were introduced), and its large size (it failed the "pocket test" by not fitting in an average coat, shirt, or pants pocket). Critics also panned its handwriting recognition. These initial problems marred Newton's reputation in the eyes of the public, and PDAs would remain a niche product until Palm, Inc. introduced the Palm Pilot, before the Newton was discontinued. The Palm Pilot, with its smaller, thinner shape; cheaper cost; and more reliable (though less intuitive) Graffiti® handwriting recognition system, managed to restore the viability of the PDA market after Newton's commercial failure.

The Newton marketing campaign trumpeted its handwriting recognition, though in initial versions it was fairly inaccurate. The original handwriting recognition engine was called Calligrapher, and was licensed from a Russian company called Paragraph International. It was actually quite sophisticated; unlike the later Palm Pilot's Graffiti which made the user learn a new handwriting system and write each letter in an input area, Newton learned the user's handwriting (using a database of known words to make guesses as to what the user was writing) and could interpret writing anywhere on the screen. Newton could also recognize and clean up simple drawn shapes such as triangles, circles, and squares, and had an intuitive system for handwritten editing (such as scratching out words to be deleted, circling text to be selected, or using written carets to mark inserts).

Later releases of the Newton operating system retained the original recognizer for compatibility, but added a printed-text recognizer, code-named "Rosetta," which was developed by Apple, included in version 2.0 of the Newton operating system, and refined in Newton 2.1. Rosetta was generally considered a significant improvement and many users consider the Newton 2.1 handwriting recognition software better than any of the alternatives since. By the time Apple discontinued the Newton in 1998, the handwriting recognition was greatly improved, and may be the best "real handwriting" recognition (as opposed to pseudo-handwriting input mechanisms like Graffiti) to have ever been available to the public. This may be one reason why the Newton still has a hard-core following to this day.

Recognition and computation of handwritten horizontal and vertical formulas such as "1 + 2 =" was also under development but never released because the principal engineer working on it went on leave.

Even given the age of the hardware and software, Newtons still demand a sale price on the used market far greater than that of PDAs produced by other companies. As of 2004 the Newton 2000 and 2100 still sell, without accessories, for over $100, despite the hardware being at least six years old.

Technical details

Newton used an advanced object-oriented programming system called NewtonScript, developed by Apple employee Walter Smith [1] (http://wsmith.best.vwh.net/). One of the major complaints programmers had was that the Toolbox programming environment was overpriced at $1000 (late in the life of the Newton the programming environment was made available for free). Additionally, it required learning a new way of programming. Despite this, many third party and shareware applications were (and continue to be) available for Newton. It has been suggested that the NewtonScript programming system should be made available open-source (as "abandonware") but most Newton enthusiasts consider this possibility to be highly unlikely.

Data in Newton was stored in object-oriented databases known as *soups*. One of the revolutionary aspects of Newton was that soups were available to all programs; and programs could operate cross-soup; meaning that the calendar could refer to names in the address book; a note in the notepad could be converted to an appointment, and so forth; and the soups could be programmer-extended - a new address book enhancement could be built on the data from the existing address book.

While the soup concept worked remarkably well within the Newton system itself, it caused several usability issues. First, it made it extremely difficult to synchronize data with other systems, like a desktop Macintosh or PC, making the Newton a data island. Apple's utility to perform this task, the Newton Connection Utility, was exceedingly complex and was never completed to perform to the satisfaction of most users. The realization that a handheld computer needed to work within the existing data environment of its users was key to the success of the later Palm Pilot platform, even though the Palm was technically inferior.

The second consequence of the data-object soup was that objects could extend built-in applications such as the address book so seamlessly that Newton users could not distinguish which program or add-on object was responsible for the various features on their own system. A user rebuilding their system after extended usage might find themselves unable to manually restore their system to the same functionality because some long-forgotten downloaded extension was missing. There was no easy way to get a listing of all installed objects on a system.

Finally, the data soup concept worked well for data like addresses, which benefit from being shared cross-functionally, but it worked poorly for discrete data sets like files and documents. This difficulty in working and sharing data with other systems, stemming from the too revolutionary data-object soup system, was a key contributor to Newton's demise.

The MessagePad used Macintosh-standard serial ports (round Mini-DIN 8 connectors instead of the more common trapezoidal DE-9, commonly called DB-9. The 2000/2100 models had a proprietary small flat connector, called an InterConnect port, used with an adapter. In addition, all models also had infrared connectivity. Unlike the Palm, all MessagePad models were equipped with a standard PCMCIA expansion slot (two on the 2000/2100). This allowed native modem and even Ethernet connectivity; Newton users have also written drivers for 802.11b wireless networking cards and ATA-type flash memory cards, a category that includes the popular CompactFlash format. With the 1xx series, an optional keyboard became available, which could also be used via the dongle on a 2x00. Newton could also dial a phone number through the MessagePad speaker (simply hold a

telephone handset up to the speaker) and fax / email support was built in at the operating system level (although it required external cards).

The MessagePad 2000 and 2100, with a vastly improved handwriting recognition system, 160 MHz ARM processor, Newton 2.1, and a better, clearer, backlit screen, were among Apple's finest products. Although their large size kept them from being as popular as today's PalmOS devices, many users still swear by them. Their handwriting recognition is still considered by many the best in the world, with only the recent Tablet PC handwriting recognition system coming close. Newton 2.0 and 2.1 were in many ways a breakthrough in handheld operating systems, one that many feel has yet to be beaten even years after its discontinuation.

The MessagePad could be used with the screen turned horizontally ("landscape") as well as vertically ("portrait"). A change of a setting would instantly rotate the contents of the display by ninety degrees. Handwriting recognition would still work properly with the display rotated.

Apple and third parties marketed several "wallets" for the MessagePads, which would hold them securely along with the owner's credit cards, driver's license, business cards, and cash. These wallets were even larger than the MessagePads and even less able to fit in a pocket, so they were most often used as a protective case for the unit to shield it from bumps and scratches.

The Newton in development

The Newton project was not originally intended to produce a PDA. The PDA category did not exist for most of Newton's genesis, and the "personal digital assistant" moniker itself was coined by John Sculley relatively late in the development cycle. Newton was, in fact, intended to be a complete reinvention of personal computing. For most of its design lifecycle Newton had a large-format screen, more internal memory, and a rich object-oriented graphics kernel. One of the original motivating scenarios for the design was known as the "Architect Scenario," in which Newton's designers imagined a residential architect working quickly with a client to sketch, clean up, and interactively modify a simple two-dimensional home plan.

For a portion of the Newton's development cycle (roughly the middle third), the project's primary programming language was Dylan, a small, efficient object-oriented Lisp variant that still retains some interest as a programming language. Although it was efficient (for its day, and considering its substantial run-time dynamism), Dylan was a tough sell for the large-format Newton (and for a development team unused to Lisp programming). With the move to the smaller form factor, Dylan was relegated to experimental status in the "Bauhaus Project" and eventually cancelled outright. Had it been retained, Dylan, with garbage collection and close OS integration, would have preceded Microsoft's managed code revolution by over a decade.

The project missed by far its original goals to reinvent personal computing, and then to rewrite contemporary application programming. The Newton project's broad vision fell victim to project slippage, feature creep, and a growing fear that it would interfere with Macintosh sales. It was reinvented as a PDA which would be a complementary Macintosh peripheral instead of a stand-alone computer which might compete with the Macintosh.

eMate 300

The **eMate 300** was offered to schools in 1997 as an inexpensive ($799 US, originally sold to education markets only) and extremely durable computer for classroom use. The eMate had a 480x320 16-tone grayscale screen, a stylus, a full-sized keyboard, an infrared port, and standard Macintosh serial/LocalTalk port. Power came from built-in rechargeable batteries, which lasted 28 hours. Its exterior was a translucent plastic green shell with a built-in handle. It was supposed to be durable enough to be dropped from arm height on a hard

floor without damage, a rugged design that would eventually influence the first iBook series. In order to achieve its low price, the eMate 300 did not have all the features of the other Newton model at the time; the MessagePad 2100. The eMate had a slower ARM processor, less RAM and audio support and fewer expansion ports. Apple's wish to make the eMate 300 a low end model kept them from realizing a new market of people looking for a small ultralight laptop. The eMate 300 was cancelled along with the rest of the Newton line.

Later efforts

Many prototypes of additional Newton models were spotted. Most notable was a Newton tablet or "slate," a large, flat screen that could be written on. Others included a "Kids Newton" with side handgrips and buttons, "VideoPads" which would have incorporated a video camera and screen on their flip-top covers for two-way communications, the "Mini 2000" which would have been very similar to Palm Pilot, and the "NewtonPhone" (developed by Siemens AG) which incorporated a handset and a keyboard.

Before the Newton project was cancelled, it was "spun off" into its own company, *Newton Inc.*, but was reabsorbed several months later when Steve Jobs ousted Apple CEO Gil Amelio and reassumed control of Apple. There has since been continual speculation that Apple might release a new PDA with some Newton technology or collaborate with Palm. Apple continues to deny that such a project will ever happen.

The Apple iPod is somewhat of a descendant of the Newton in that it is a pocket-sized grayscale programmable device based on the ARM processor. Two ex-Apple Newton developers founded Pixo, the company that created the iPod's OS.

Feeding a bit of speculation, Apple put the "Print Recognizer" part of the Newton 2.1 handwriting recognition system into Mac OS X version 10.2 (known as "Jaguar"). It can be used with graphics tablets to seamlessly input handwritten printed text anywhere there was an insertion point on the screen. This technology, known as "Inkwell", appears in the System Preferences whenever a tablet input device is plugged in. Whether Apple will ever utilize such technology again in a handheld device remains to be seen.

In June 2004, Apple CEO Steve Jobs indicated that he was proud that Apple resisted pressure to market a new handheld computer. While a small group of Mac faithful consumers have lobbied Apple to sell such a device, the worldwide market for PDAs was in a decline at the time, and Apple chose not to develop the device because demand would have been inadequate.

Newton models

- MessagePad (also known as the H1000, OMP or Original MessagePad)
- MessagePad 100
- MessagePad 110 (slightly longer and narrower, with integrated flip cover and retracting stylus)
- MessagePad 120
- MessagePad 130 (backlit)
- eMate 300 (backlit with built-in keyboard)
- MessagePad 2000 (a significant upgrade; much faster, larger form factor)
- MessagePad 2100 (raised internal RAM to 4MB)

The NewtonOS was also licensed to a number of third party developers including Sharp and Motorola who

developed additional PDA devices that used the operating system.

Appearances in popular culture

- The Newton was featured in the movie *Under Siege 2,* where the main character, played by Steven Seagal, uses it to fax a call for help from a phone on a passenger train.
- Garry Trudeau ridiculed it in a series of episodes of his popular comic, *Doonesbury.* The last panel of one strip, which shows a character reading the words "egg freckles?" from his Newton [2] (http://images.ucomics.com/comics/db/1993/db930827.gif), became an Easter egg in the Newton operating system itself (version 2.0 and earlier). It can be seen by writing the words *egg freckles* then highlighting them and tapping the Assist button.
- In an episode of *The Simpsons* titled "Lisa on Ice", which first aired November 13, 1994, school bully Kearney has his buddy Dolph take a memo on a Newton. When Dolph writes "Beat up Martin" on the screen, the handwriting recognition turns it into "Eat up Martha." The bully throws his Newton at Martin instead. [3] (http://www.snpp.com/episodes/2F05.html)
- In early episodes of the series *The X-Files*, the FBI agents use Newtons.
- In the end scene of Larry Laffer *Leisure Suit Larry 6: Shape Up or Slip Out!* the woman says "I even had a Newton".
- The character of Kate Libby in Hackers has a MessagePad which is seen in a number of scenes.
- Ridicule of the handwriting recognition led to the joke: 'How many Newton users does it take to change a lightbulb? Foux. There to eat lemons, axe gravy soup.'
- The hacker in the film *Jurassic Park* has a Newton on his desk.

External links

- Newton FAQ (http://www.chuma.org/newton/faq/)
- Newton Gallery (http://www.msu.edu/~luckie/newtgal.htm)
- Larry Yaeger's page on the development (http://homepage.mac.com/larryy/larryy/ANHR.html) of the Rosetta recogniser engine
- An interview with Larry Yaeger (http://arstechnica.com/journals/apple.ars/2005/6/12/504) touching on the development of the Newton and its HWR
- Newton Secrets (http://www.a-in-a-circle.com/newton/), with photos of prototypes
- Newton Cadillac prototype info (http://www.uzes.net/newton)
- History of Newton handwriting recognition technology (http://www.chuma.org/newton/ntalk-archive/sept2000/0152.html)
- Think you know the Apple Newton's History? Think again (http://osopinion.com/modules.php?op=modload&name=News&file=article&sid=4556)
- Newton Software (http://www.unna.org/)
- The NewtonTalk mailing list (http://www.newtontalk.net/)
- NewtonSearch (http://www.newtonsearch.net/), a searchable index of Newton websites
- NewtonRepair (http://www.angelfire.com/oz/newtonian/), Apple has discontinued support for the Newton platform however repairs and upgrades are still available at this site
- Newton programming books and references in PDF form (http://www.pda-soft.de/programmingbooks.html)

Retrieved from "http://en.wikipedia.org/wiki/Apple_Newton"

Categories: Failed Apple initiatives | Apple hardware | PDAs

Dylan programming language

From Wikipedia, the free encyclopedia.

Dylan is a dynamic programming language created by a group led by Apple Computer. It was originally intended for use with Apple's Newton computer, but their implementation did not reach sufficient maturity in time, and they instead developed NewtonScript for that project. A "technology demonstration" version for writing Macintosh applications was released in 1995, based on an advanced IDE, but by this time Apple had already publicly abandoned Dylan, and developers avoided it even at the $29 price. The language design was intriguing enough that two other groups developed optimizing compilers for Dylan: Harlequin Inc. (now Functional Objects) released a commercial IDE for Microsoft Windows, and Carnegie Mellon University released an open source compiler for Unix systems. Both implementations are now being maintained and extended by a group of volunteers as Gwydion Dylan.

Dylan is essentially a cleaned-up version of CLOS, an object-oriented programming system built on Common Lisp. In Dylan, almost all entities (including primitive data types, methods, and classes) are first-class objects. One tremendous advantage of Lisp-like languages is that nearly every component of the system, including the actual language itself, can be modified from within the language. This makes Lisp systems incredibly flexible. However many programmers have been turned off by Lisp's seeming odd and unfamiliar syntax. Another issue is that early Lisp systems, because of their dynamism and flexibility, did not always perform as well in some applications as static programming languages on the limited hardware of the day. For various reasons, Lisp did not enjoy widespread use for developing commercial software.

Dylan's main design goal was to be a dynamic language well-suited for developing commercial software. Dylan attempted to address the performance problem by introducing "natural" limits to the full flexibility of Lisp systems, allowing the compiler to clearly understand compilable units. Early versions of Dylan were otherwise similar to existing CLOS systems, but developer feedback in the 1993 era forced them to send the product back into engineering and produce a clearer syntax as well.

Contents

Syntax

At first, Dylan used Lisp syntax, which is based on parenthesis:

```
(bind ((radius 5)
       (circumference (* 2 $pi radius))
  (if (> circumference 42)
      (format-out "Hello big circle! c is %=" circumference)
    (format-out "Hello circle! c is %=" circumference)))
```

The language was then changed to use an Algol-style syntax, which would be more familiar to C programmers:

```
begin
 let radius = 5;
 let circumference = 2 * $pi *radius;
 if (circumference > 42)
    format-out("Hello, big circle! c = %=", circumference);
 else
   format-out("Hello, circle! c is %=", circumference);
end;
```

Modules vs. namespace

In most OO languages the concept of *class* is the primary encapsulation system; the language is generally thought of as "a way to make classes". Modern OO languages often also include a higher level construct known as the *namespace* in order to collect related classes together. In addition the namespace/class system in most languages defines a single unit that must be used as a whole, if you want to use the String.concat function, you must import and compile against all of String, or the namespace that includes it.

In Dylan the concepts of compile-unit and import-unit are separated, and classes have nothing specifically to do with either. A *module* defines items that should be compiled and handled together, while an *interface* defines the namespace. Classes can be placed together in modules, or cut across them, as the programmer wishes. Often the complete definition for a class does not exist in a single module, but is spread across several that are optionally collected together. Different programs can have different definitions of the same class, including only what they need.

What's the difference? Well consider an add-on library for regex support on String. Under traditional languages in order for the functionality to be included in strings, the functionality has to be added to the String namespace itself. As soon as you do this, the String class becomes larger, and people who don't need to use regex still have to "pay" for it in increased library size. For this reason these sorts of add-ons are typically placed in their own namespaces and objects. The downside to this approach is that the new functionality is no longer a *part of* string, instead it is isolated in its own set of functions that have to be called separately. Instead of the clean `myString.parseWith(myPattern)` syntax that follows classical OO concepts, you are forced to use something like `myPattern.parseString(myString)`, which effectively reverses the natural ordering.

In addition, under Dylan many interfaces can be defined for the same code, for instance the String.concat could be placed in both the String interface, and the "concat" interface which collects together all of the different concatenation functions from various classes. This is more commonly used in math libraries, where functions tend to be applicable to widely differing object types.

A more practical use of the interface construct is to build public and private versions of a module, something that other languages include as a "bolt on" feature that invariably causes problems and adds syntax. Under Dylan the programmer can simply place every function call in the "Private" or "Development" interface, and collect up publicly accessible functions in "Public". Under Java or C++ the visibility of an object is defined in the code itself, meaning that to support a similar change the programmer would be forced to re-write the definitions completely, and could not have two versions at the same time.

Classes

Classes in Dylan describe "slots" (data members, fields, ivars, etc.) of objects in a fashion similar to most OO languages. All access to slots are via methods, a feature of most dynamic languages. Default getter and setter methods are automatically generated based on the slot names. In contrast with most other OO languages, other

methods applicable to the class are often defined outside of the class, and thus class definitions in Dylan typically include the definition of the storage only. For instance:

```
define class <window> (<view>)
  slot title :: <string> = "untitled", init-keyword: title:;
  slot position :: <point>, required-init-keyword: position:;
end class;
```

In this example the class "window" is created. The <class name> syntax is convention only, to make the class names stand out. In most languages the convention is to capitalize the first letter of the class name instead, or add additional characters. Window inherits from a single class, `<view>`, and contains two slots, `title` holding a string for the title at the top of the window, and `position` holding an X-Y point for the upper corner of the window. In this particular example the title has been given a default value, while the position has not. The optional "init-keyword" syntax allows the programmer to specify the initial value of the slot when instantiating an object of the class.

In languages such as C++ or Java, the class would also define its interface. In this case the definition above has no explicit instructions, so in both languages access to the slots and methods is considered `protected`, meaning they can be used only by subclasses. In order to allow unrelated code to use the window instances, they would have to be declared `public`. In Dylan these sorts of visibility rules are not considered part of the code itself, but of the module/interface system. This adds considerable flexibility. For instance, one interface used during early development could declare everything public, whereas one used in testing and deployment could limit this. With C++ or Java these changes would require changes to the source code itself, so people don't do it, whereas in Dylan this completely unrelated concept is completely unrelated.

Although this example does not use it, Dylan also supports multiple inheritance. The developers spent enough time on the classloader to avoid the problems that continue to make many uninformed programmers believe that multiple inheritance is a "bad idea".

Methods and generic functions

In Dylan, methods are not intrinsically associated with any particular class; methods can be thought of as existing outside of classes. Like CLOS, Dylan is based on multimethods, where the specific method to be called is chosen based upon the types of all its arguments. The method does not have to be known at compile time, the understanding being that the required functionality may be available or may not, based on the user's preferences.

Under Java the same methods would be isolated in a particular class. In order to use that functionality the programmer is forced to *import* that class and refer to it explicitly in order to call the method. If that class is not available, or unknown at compile time, the application simply won't compile.

In Dylan code is isolated from storage in *functions*. Many classes have methods that call their own functions, thereby looking and feeling like most other OO languages. However code may also be located *generic functions*, meaning they are not attached to a particular class, and can be called natively by anyone. Linking a particular generic function to a method in a class is accomplished this way:

```
define method turn-blue (w :: <window>)
  w.color := $blue;
end method;
```

This definition is similar to those in other languages, and would likely be encapsulated within the <window>

class. Note the := setter call, which is syntactic sugar for `color-setter($blue, w)`.

The utility of generic methods comes into its own when you consider more "generic" examples. For instance, one common function in most languages is the `to-string`, which returns some human-readable form for the object. For instance, a window might return its title and its position in parens, while a string would return itself. In Dylan these methods could all be collected into a single module called "to-string", thereby removing this code from the definition of the class itself. If a particular object did not support a to-string, it could be easily added in the to-string module.

Extensibility

This whole concept might strike some readers as very odd. The code to handle to-string for a window isn't defined in <window>? This might not make any sense until you consider how Dylan handles the call to to-string. In most languages when the program is compiled the to-string for <window> is looked up and replaced with a pointer (more or less) to the method. In Dylan this occurs when the program is first run instead, the runtime builds a table of method-name/parameters details and looks up methods dynamically via this table. That means that a function for a particular method can be located anywhere, not just in the compile-time unit. In the end the programmer is given considerable flexibility in terms of where to place their code, collecting it along class lines where appropriate, and functional lines where it's not.

The implication here is that a programmer can add functionality to existing classes by defining functions in a separate file. For instance, you might wish to add spell checking to all <string>s, which in most languages would require access to the source code of the string class -- and such basic classes are rarely given out in source form. In Dylan (and other "extensible languages") the spell checking method could be added in the `SpellCheck` module, defining all of the classes on which it can be applied via the `define method` construct. In this case the actual functionality might be defined in a single generic function, which takes a string and returns the errors. When the SpellCheck module is compiled into your program, all strings (and other objects) will get the added functionality.

This still might not sound all that obvious, but in fact it is a common problem faced by almost all OO languages; not everything fits into a class construct, many problems apply to *all* objects in the system and there's no natural way to handle this. A concrete example of this flexibility is what most other languages refer to as aspect oriented programming, where a particular function "cuts across" most of the objects in the system. This functionality cannot be provided in the basic language, leading to a series of add-ons and non-standard versions of the language as they invariably attempt to add it at some later date. For instance Java is currently attempting to add this sort of system in AspectJ, but this is proving very difficult and many Java programmers question the need for it in the first place. Suffice it to say, once you have used a language that allows extensions, you'll never want to go back.

External links

- Gwydion Dylan (http://www.gwydiondylan.org/) - host of two optimizing Dylan compilers targeting Unix/linux, Mac OS X, and Microsoft Windows
- Functional Objects (http://www.functionalobjects.com/) - developers of the Harlequin Dylan compiler for Microsoft Windows
- Dylan Language Wiki (http://monday.sourceforge.net/wiki/)
- Dylan Language White Paper (http://www.functionalobjects.com/resources/white-paper.phtml)
- Open Directory Project: Dylan (http://www.dmoz.org/Computers/Programming/Languages/Dylan/)
- Harlequin (now Global Graphics) (http://www.harlequin.com/) - former developers of commercial Dylan, Lisp, and ML compilers
- More Dylan programming language links (http://www.coderlinks.de/dylan,1200,1201.html)

Retrieved from "http://en.wikipedia.org/wiki/Dylan_programming_language"

Categories: Programming languages | Failed Apple initiatives

Home
Products
Support
News
About Dylan
Advantages
Benefits
• **White Paper**
COM Simplified
Interactive CORBA
Beyond Java
Links
About Fun'o
contact Fun'o

Dylan Language White Paper

Version 1.01; 20 January 1998, modified October 1, 1999

This white paper describes the general structure and specific features of the Dylan programming language. Rather than being a tutorial, the paper provides an overview of the language and rationale for some aspects of the design.

Feedback on the current contents and proposed extensions is welcome. Comments can be sent to dylan-support@functionalobjects.com.

Background

The Dylan programming language was originally developed at Apple Computer, in cooperation with researchers from Harlequin Group plc, Carnegie Mellon University, and elsewhere. These groups saw the need for a new language, one which combined the efficiency of C++ with the simplicity of Smalltalk. They felt that such a language---with native compilation, automatic memory management, and a range of other carefully designed features---would enable a new generation of software development.

Unfortunately, Apple fell on hard times just as its design and initial implementation of Dylan were being completed. Apple shipped one version of their Dylan IDE prototype in 1995, and this version was enthusiastically received by developers. Shortly after the release, though, Apple abandoned their Dylan effort during an overall reduction in research and development.

The end of Dylan at Apple did not, however, mark the end of Dylan. Language development continued at Harlequin, which had begun work on a version for the Wintel platforms. The first version of Harlequin Dylan was released in July 1998. In 1999, Harlequin fell on hard times, and decided to spin off its Dylan assets in order to concentrate its own efforts in other areas. Functional Objects, Inc. was thus formed to take Dylan development to the next level.

Design Goals

From the outset, Dylan was designed to be a general purpose, fully object-oriented language for use in systems, application and

component programming. The language was designed to be both simple and powerful, internally consistent and feature rich.

Dylan is both a high-level and a low-level language. It is high-level in that programs are easy to write, easy to read, and easy to extend. The language provides type safety and garbage collection, and is thoroughly object-oriented. At the same time, Dylan is low-level in that it can access native platform APIs, it can be compiled efficiently and it can therefore be used for the full range of programming tasks.

Beyond all else, Dylan was designed to help programmers create efficient abstractions. An efficient abstraction is one that allows programmers to think about their programs efficiently and that simultaneously allows the compiler to generate efficient compiled code. Dylan shows that you don't need to sacrifice clean code in the name of performance.

It is Dylan's ability to support efficient abstractions that most distinguishes it from other languages. This support makes Dylan programs substantially easier to write, maintain, and extend, without sacrificing performance.

Fully Generalized

Dylan avoids special cases and artificial constructs. Instead, it provides clean general mechanisms that can be used for a variety of purposes. This approach makes the language easier to learn, and ultimately more powerful.

The following list shows some of the areas in which Dylan provides a general mechanism that can be used to mimic the special cases of other languages, but that can also support other uses:

- Many other languages support only single inheritance. Dylan supports multiple inheritance, but without sacrificing the efficiency of single inheritance when it is important.
- Most other object-oriented languages dispatch only on the type of one argument (i.e. the receiver, sometimes known as *this*). Dylan can dispatch on the type of one argument, or more than one argument.
- Where Java and C++ support public, private, and protected variables, Dylan allows the creation of multiple interfaces, each containing exactly those objects determined to be appropriate for a given client.
- In many other languages, classes are the unit of encapsulation. In Dylan, classes can be written to follow encapsulation boundaries or, alternatively, encapsulation

boundaries can cut across class lines. The latter course often allows more natural implementation sharing and more precise information hiding.
- In some other object-oriented languages, procedural programming is deprecated, forcing programmers to use constructs that are artificially object-oriented, and therefore awkward. Dylan integrates object-oriented and procedural programming, allowing programmers to use the best tool for the job, all within a unified framework.
- In many other languages, some data are objects and others are not. In Dylan, all data are objects and fully participate in the object system.

The Dylan Object System

When we say that Dylan is a fully object-oriented language, we mean that all data is represented as instances, and all instances are treated equally. The same method dispatch mechanism is applicable to all instances---including classes, functions, and numbers---and all instances can be stored in variables, passed as function arguments, and returned as function results. In contrast, many other languages have some data types that must be handled as special cases or be cast to object types before they can participate in the object system.

While it is thoroughly object-oriented, Dylan's approach to objects differs from that of most other object-oriented languages. Dylan is not a class-centric language. Programs are not written inside classes, methods are not stored inside classes, and slots (instance variables) are not referenced as variables scoped inside instances.

Instead, Dylan provides object-orientation within a clean lexically scoped framework. Programs are written in modules. Classes and generic functions appear as named constants in these modules. Methods are stored in generic functions, and method dispatch occurs when a generic function is called. Slots are accessed abstractly through methods in generic functions.

A module may contain many class definitions and many generic function definitions, and these two kinds of objects are both named at top level in the module.

This approach gives the programmer complete control over program structure. Programmers are not forced to artificially associate functions with a single class when it is not appropriate to do so. Encapsulation boundaries are not forced to follow class boundaries. Closely related classes can share private interfaces in a single implementation module, or conversely, the

implementation of a class may be split across modules. In general, information can be appropriately shared or hidden within a complex implementation, without externally imposed structural constraints.

Modules

Dylan programs are constructed in namespaces called modules. A module is simply a compile time mapping from names to values (objects) or value-locations (variables). As described above, these values include the classes, functions, and other instances that form the substance of a Dylan program.

A module may export names, and thereby export the objects they indicate, and it may import names from other modules. Imported names may be reexported. Because the names are nothing more than compile time mappings, Dylan can support flexible compile time renaming to prevent any possibility of irreconcilable name clashes, even when using modules for which the programmer does not have source code.

A module may contain zero or more class definitions, zero or more function definitions, and other object definitions. Reduced to the idiom enforced by other languages, a module would contain a single class definition and all the methods associated with that class. Most Dylan programmers do not, however, choose to program in this idiom because they find it an unnatural way to decompose problems. It practice, it is more common to use modules that import many objects (including classes and functions) from other modules, implement several new objects, and then export an interface consisting of some of those newly created objects and perhaps some of the imported objects.

Programmers have complete control over the selection of names to be exported from a module. Because classes and functions are accessed through names, this translates to complete control over the interface that is visible to clients of the module.

By importing and selectively re-exporting names, modules can effectively provide multiple interfaces to a given implementation. Each interface can include an arbitrary set of exported names. This lets the programmer give clients access to the functionality that is precisely appropriate for their use. For example, one interface could be used by subclassers, another for clients who wish to survive version changes, and another for clients who are willing to sacrifice compatibility over version changes in exchange for access to power-user features.

The access control provided by the Dylan module system can be

used to mimic the access control provided by Java and C++. The latter languages give you three special cases to handle common needs: public (full outside access), private (no outside access), and protected (outside access only by subclassers). While Dylan modules can be used to support this factoring strategy, it can also be used to support other factoring strategies.

Classes

Classes define the structure of Dylan instances and categorize instances by type. Dylan classes are not scopes for names nor are they containers of methods.

Dylan classes describe their instances. They describe the storage each instance provides, how that storage is initialized, and where the instance lies in the type hierarchy. The behavior of instances is specified by methods, and access to objects (including classes and generic functions) is controlled by the module system. By keeping these areas orthogonal, Dylan allows programs to be structured very precisely.

The following example defines a new class with one direct superclass and two slots.

```
  define class <window> (<view>)
    slot title :: <string> = "untitled", init-keyword:
title:;
    slot position :: <point>, init-keyword: position:;
  end class <window>;
```

This example defines a new class, `<window>`, which inherits from the class `<view>`. By convention, angle brackets are placed around class names to make classes more visually distinct in source code, and to reduce accidental name collisions. The angle brackets are part of the name of the class, and are not special syntax.

The `title` slot is constrained to hold values of type `<string>`. The initial value of this slot can be specified by the `title:` keyword argument when a fresh instance is created. If the keyword argument is not specified, the slot is initialized to the default value `"untitled"`.

The `position` slot is constrained to hold values of type `<point>`. The initial value of this slot can be specified by the `position:` keyword argument; otherwise there is no default initial value and the slot will remain uninitialized until it is explicitly set. Attempting to retrieve the value of an uninitialized slot signals an error. This behavior eliminates the possibility of confusing a system-supplied

default value with a valid value in a partially initialized instance. Slots only have a default value when they are so defined.

(While it is somewhat unrealistic not to specify a default initial value for a window position, there are many cases where a meaningful initial value cannot be computed until after the instance is created. It is in those cases where it is especially important that programs not accidentally use a bogus default value.)

The class defined above will be stored as the value of the module constant `<window>`. If the module exports the name `<window>`, clients of the module will have access to the class. Otherwise, clients will not have access to the class. Similarly, the slot accessors (getters and setters) will be stored as the values of the module constants `title`, `title-setter`, `position`, and `position-setter`. Clients of the module will have the ability to get and set the values of the slots only to the extent these names are exported.

The syntax for using slot accessors is described in a separate section.

Functions

Dylan includes two kinds of functions: methods and generic functions. Both of these kinds of functions may be called directly. More commonly, methods are collected in generic functions. When a generic function is called, it compares its arguments to the methods it contains, chooses the best match, and then invokes that method. This dispatch will be performed at compile time or run time, depending on the compile time information available. Generic functions contain no code of their own; they only contain methods.

The following example defines two methods on the `contents` generic function:

```
define method contents (w :: <window>)
  bitmap(w); // return the window's bitmap
end method contents;

define method contents (b :: <cracker-jack-box>)
  toy-surprise(b); // return a toy surprise
end method contents;
```

Once these methods have been defined, the `contents` function can be called:

```
contents(some-object)
```

If `some-object` is an instance of `<window>`, the first method will be invoked. If it is an instance of `<cracker-jack-box>`, the second method will be invoked.

An alternative syntax is provided for calling functions of one argument:

```
some-object.contents
```

This call is semantically equivalent to the previous one. They differ only in syntax. In both cases, the generic function `contents` is called with the argument `some-object`, and the type of that argument is used to select the appropriate method. Some programmers prefer the dot syntax when calling a function that conceptually retrieves or sets a property of an object.

To set the `contents` of an object, one could use any of the following expressions. All are equivalent, and all depend on the existence of an appropriate method definition for `contents-setter`. (Note that the first form would not generally be used directly. It is shown here to illustrate the underlying expression for which the other two provide convenient syntax.)

```
contents-setter(new-contents, some-object)
contents(some-object) := new-contents
some-object.contents := new-contents
```

Multimethods

The generic function `contents` accepts only one argument, so it is the type of that argument that determines the method to be invoked. When a generic function accepts more than one argument, any number of the arguments can be used to affect the dispatch. If the available methods differ only in the type of one argument, the semantics are the same as in languages with an implicit receiver, such as C++ and Java. However, there are many cases where dispatching on the types of multiple arguments can be very useful.

For example, the `contents-setter` method for windows might be defined as follows:

```
  define method contents-setter (new-value :: <bitmap>,
w :: <window>)
    w.bitmap := new-value;
  end method contents-setter;
```

Note that the *new-value* argument is specified to have the type `<bitmap>`, perhaps because this is the type required by the `bitmap-setter` function, or perhaps because at the time the

method was written, bitmaps were the only type of graphic object available.

But what if we want `contents-setter` to be more general than `bitmap-setter`? What if we have recently gotten a gif library with a `gif-to-bitmap` function, and we want `contents-setter` to support gif images?

In other languages, this might be done by editing the original definition of `contents-setter`, and including conditionals to handle the different types of *new-value*. In Dylan, the same result can be accomplished without editing the source code of the original method, and without using explicit (non-object-oriented) conditionals. We do this by adding another method to `contents-setter`.

```
  define method contents-setter (new-value :: <gif-image>,
w :: <window>)
    w.contents := gif-to-bitmap(new-value);
  end method contents-setter;
```

Now if `contents-setter` is called with a gif image and a window, the gif image is converted to a bitmap, and `contents-setter` is reinvoked with the new values. The complete call chain would be

`contents-setter` generic function
dispatches to `contents-setter(<gif-image> <window>)` method
converts *new-value* from a gif to a bitmap
and then calls `contents-setter` generic function again
dispatches to `contents-setter(<bitmap> <window>)` method

This type of coercion is a typical example of how multimethods can be used to factor common programming problems very naturally. The example also shows how protocols can be extended to support new classes, without editing the source code of the original protocol.

Implications of Generic Functions

It should be clear by now that Dylan methods do not have an implicit *this* parameter. All the parameters are explicit, and all are equally important in determining the outcome of method dispatch.

A related fact is that method definitions are not placed within the text of class definitions. They are simply defined in modules, alongside other object definitions. This means that new protocols can be defined on existing classes without modifying the source

code of those classes. For example, a programmer creating a WYSIWYG interface designer could set up event handlers and define new generic functions on the standard widget classes, thereby supporting live editing of the properties of the widgets. This could be done without modifying the source code for the underlying widget library. The widget classes would thus be extended without being modified internally.

Finally, as noted above, generic functions are objects just like any other. The method definitions shown above first ensure that the module constants `contents` and `contents-setter` exist, and that they contain the appropriate generic functions. The definitions then add the specified methods to those generic functions. The programmer can expose the generic functions (and hence the methods) to clients by exporting the names `contents` and/or `contents-setter`, or the programmer can keep them hidden by not exporting the names.

Other features of Dylan functions are described in later sections of this document.

Slots

When a class is defined, methods are automatically defined to access slots in instances of the class. The slots are then accessed by making standard function calls to the generic functions containing these methods.

There is no distinguished view of slots from "inside" a class. Slots are accessed the same way by implementers and clients: always through a standard function call.

For example, the position and the contents of a window class would be accessed the same way.

```
position(my-window) or my-window.position
contents(my-window) or my-window.contents
```

It is an implementation detail that one of these methods is implemented as a slot while the other is a more complex method. This fact about the implementation is hidden from code which uses `contents` and `position`. The hiding allows the implementations to be changed during the development or revision process without requiring changes to clients or to other parts of the implementation of the `<window>` class. The only things that need to change are the slot portion of the class definition and the single method definition.

A single generic function may contain both slot accessors and

other method definitions. For example, while `<window>` implements `position` as a slot, another class could define an explicit method on `position`, and this method could perform any calculation required to compute the appropriate result.

When a slot is defined, both the getter and setter methods are defined. Each has its own name, and each can be exported independently. In Java, it is common to keep instance variables private and provide public access to them through methods which provide a more abstract interface. In Dylan, this extra step is not needed. The basic slot access mechanisms are already abstract, and provide sufficient abstraction and access control.

It is important to note that while slot accessors have the syntax and semantics of function calls, they are not compiled as function calls. The Dylan compiler can often detect the nature of the operation at compile time and create an in-line slot access, just as it would if the slot were accessed using special syntax.

Macros

Dylan supports a hygienic pattern-template based macro system. Rather than performing text-to-text transformation, the macro system manipulates syntactic patterns. This approach has several advantages. It allows programmers to construct many standard Dylan forms, including definitions and statements, and it guarantees that macro calls will maintain the syntactic flavor of the language. It also avoids the accidental name clashes and name captures typical in simpler macro systems, regardless of where the macro is written and where it is used.

Macros are commonly used to customize the language so that it directly and declaratively expresses the structure of the problem domain for a given project. For example, it is quite common to create additional defining forms (e.g. `define graphics-frame`) to extend the built in set of defining forms (e.g. `define class`, `define method`, etc.).

Macros are a powerful tool for hiding complexity, creating layered systems, and ensuring full compliance with protocols. Macros can also be used to divide a project among system architects (those who define the macros) and system programmers (those who use the macros).

For example, the following macro ensures that COM objects used by a Dylan program have their reference counts appropriately incremented and decremented, even in the presence of errors.

```
define macro with-interface
```

```
    {
     with-interface (?variable-name:name = ?
interface:expression)
       ?block-body:body
     end
    }=> {
     let ?variable-name = ?interface;
     block ()
       Increment(?variable-name);
       ?block-body;
     cleanup // the cleanup clause is guaranteed to run
       Decrement(?variable-name); // regardless of how the
body terminates.
     end block;
    }
  end macro with-interface;
```

A call to this macro would take the following form:

```
  with-interface (request = sc.IScriptingContext-Request)
    // Code using request goes in here.
    do-something (request)
  end with-interface;
```

The call would expand to the following Dylan code:

```
  let request = sc.IScriptingContext-Request;
  block ()
    Increment(request);
    // Code using request code in here.
    do-something (request);
  cleanup
    Release(request);
  end block;
```

Sealing

Sealing is the term used to describe a number of facilities in Dylan that allow a programmer to describe more precisely the manner in which they will be using a class, method, or generic function. In particular, they describe whether a class can be given additional subclasses, how the class can be used in multiple inheritance, and whether methods can be overridden. These descriptions increase code readability, enforce programmer intent, and assist in the production of more efficient object code.

The simplest example of sealing is the sealed class directive. A sealed class cannot be given any subclasses beyond those in the library containing the class's definition. In other words, a sealed class may have subclasses, but all of its subclasses will be in the same library, will be known at compile time, and will also be sealed. This is in contrast to an open class, which can be subclassed in client libraries.

(In comparison, final classes in Java cannot have any subclasses. The utility of subclassing a sealed class is illustrated by an abstract superclass with several concrete subclasses. All these classes can be sealed and exported. The abstract superclass may be used for type declarations and dispatching while the concrete subclasses are used for implementation. Although the abstract superclass has been exported, the programmer and compiler do not need to worry about it being further subclassed.)

A less restrictive form of class sealing is the primary class declaration. Primary classes cannot be used as mixins with other primary classes, and the subclasses of a primary class are primary. This effectively creates a single inheritance chain among primary classes, ensuring that a slot defined by a primary class can always appear at the same location in all instances of the class and its subclasses. (Because a class will never have multiple primary superclasses, there will never be contention for whose slots should appear first. Slots are simply added in the order in which the classes are defined.) Combined with other type inference information, primary classes allow the slot access to be compiled as a simple indexed load instruction, rather than the more complex table lookup that would otherwise be required. In this way, primary classes allow Dylan to support multiple inheritance in general while retaining the efficiency of single inheritance when the full power of multiple inheritance is not needed.

Finally, generic functions can be sealed for portions of the type hierarchy. This prevents overriding the methods that are applicable to those types. No method can be added to the generic function that would be more specific than the existing methods. Important special cases include individual sealed methods and sealed slots.

(The relationship between sealed generic functions and Java's final methods is analogous to the relationship between sealed and final classes. In both cases, Java lets you fix leaves of a tree, where Dylan lets you fix both branches and leaves.)

Taken together, these sealing declarations often allow object dispatch to be performed at compile time rather than run time. Thus, Dylan can recapture much of the efficiency of a static procedural language while preserving the semantic clarity of a purely object-oriented dynamic language. Equally important is that sealing increases program predictability by enforcing restrictions on how objects can be used.

Exception Handling

Dylan provides a signaling exception system that is object-based and supports both termination and recovery.

When an error or other exceptional situation is discovered, it is signaled by the code that made the discovery. Dynamically installed handlers of the appropriate type have the opportunity to address (handle) the exception. If all decline, the exception is handled by a default handler, usually a debugger or fatal error message.

Exceptions are objects, instances of the `<exception>` class and its subclasses. Handlers are methods. The matching of exceptions to handlers follows the same rules as in normal method dispatch, except that instead of searching for applicable methods in a generic function, the search is performed on the call stack. The prioritization of applicable methods also differs. In generic functions, the method which most closely matches the argument types is given priority. In the exception system, the handler that was most recently installed is given priority.

Because exceptions are objects, they can be given slots to hold additional information about the error, and they can be given methods to support additional behavior, such as formatting the text of an error message.

Handlers are located, prioritized, and called without unwinding the stack. This gives the handler access to the full dynamic state of the program at the point at which the exception occurred. It also gives the handler the choice of resuming execution after repairing the problem or aborting execution by performing a non-local exit to the point at which it was installed.

The Dylan exception system is not *checked* in the Java sense of the term. While all exceptions are guaranteed to be noticed (either by an installed handler, or by a default handler), functions do not need to declare every exception they might signal directly or indirectly. This means that unlike in Java, condition handling in Dylan cannot generally be analyzed at compile time. In return, there is a great deal less bookkeeping required when using functions which signal exceptions.

Optional Arguments

In addition to receiving required arguments, Dylan functions can receive two kinds of optional arguments: keyword arguments and rest arguments.

Keyword arguments are used to specify optional arguments by name. If a particular keyword argument is not supplied in a

function call, it will be set to a specified default value. If a function accepts more than one keyword argument, the keyword arguments may be supplied in any order and in any combination.

```
define method get-sundae (#key flavor = $vanilla,
                               topping = $whipped-cream,
                               mixins = $marshmallows)
  // code to make and deliver a sundae goes here
end method get-sundae;

// just use the defaults
get-sundae()

// chocolate ice cream, no mixins, default topping
get-sundae(flavor: $chocolate, mixins: #f)

// coffee ice cream, almonds, no topping
get-sundae(mixins: $almond, flavor: $coffee, topping: #f)
```

In addition to their other uses, keyword arguments are very handy for initializing new instances. Class definitions can automatically generate keyword arguments to initialize slots in new instances. For example, given the definition of `<window>` shown above, a new window could be created using any of the following calls. These calls are all supported by the class definition. No additional constructors need to be defined.

```
make(<window>)
make(<window>, title: "New")
make(<window>, position: point(0, 0))
make(<window>, position: point(0, 0), title: "New")
make(<window>, title: "New", position: point(0, 0))
```

Rest arguments are used to collect any number of additional arguments after the required arguments have been supplied; the additional arguments are stored in a collection which is passed as the value of the rest argument. If no additional arguments are supplied, the collection is initially empty.

For example, the following function could be used to close several windows at once:

```
define method close-several (#rest windows)
  for (w in windows)
    close(w);
  end for;
end method close-several;
```

Multiple Values

Just as a function can accept multiple arguments when it is called, Dylan functions and other expressions can return multiple values.

For example, the Dylan function `truncate/` performs integer division, returning both the quotient and remainder.

```
truncate/(105, 25) => 4, 5
```

Multiple values are useful when a single computation naturally produces multiple useful results. Rather than performing the same computation multiple times or allocating storage for the results, the results can all be returned directly.

First Class Functions

Functions, including methods and generic functions, are first class objects. This means that they can be passed as arguments to other functions, stored in variables, collections and slots, and returned as the results of expressions.

An alternative definition of `close-several` provides a good example of the use of a functional argument.

```
define method close-several (#rest windows)
  do(close, windows);
end method close-several;
```

The function `do` takes two arguments: a function and a collection. It then applies the function to each element of the collection. This achieves the same behavior as that of the previous definition, which used a loop. But the new version is shorter, and some would argue easier to understand.

Another example of functional arguments is the function `choose`, which takes a predicate function as its first argument, and a collection as its second argument. It returns a new collection containing only those elements of the argument collection that pass the predicate.

```
choose(even?, #(1, 2, 3, 4, 5, 6, 7, 8, 9));
  => #(2, 4, 6, 8)
```

Again, this code could be written a loop which explicitly tested each element and collected the result by hand, but such a loop would be far more complicated to write and to read than the simple call to `choose`.

Additional Topics

The following additional topics are being considered for future drafts of this white paper.

- Syntax
- Initialization protocol
- Delegated Instantiation
- Multiple inheritance
- Constant slots
- Inherited slots
- Loose typing
- First-class types
- Non-class types (i.e. union types, singletons, subclass types, limited types)
- Unbound slots compared to the use of null
- More muli-method examples
- Closures
- Function composition
- Function application
- Tail-call optimization
- Collections
- Iteration protocol
- Subclassing `<number>`
- Optimizing numeric operations
- Control constructs and looping
- First class non-local transfers
- Protected regions

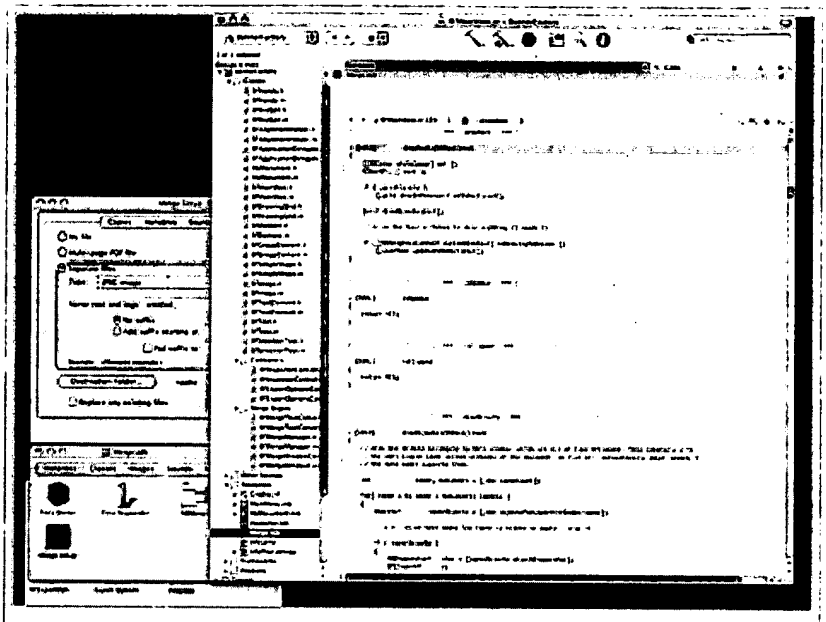
DELIVER YOUR DREAMS

Please mail webmaster@functionalobjects.com with questions or comments about this website.

Cocoa (API)

From Wikipedia, the free encyclopedia.
(Redirected from Cocoa (software))

Cocoa is Apple Computer's native object-oriented application programming environment for the Mac OS X operating system. It is one of five major programming environments available for Mac OS X; the others are Carbon, Classic, BSD, and Java. (Environments such as Perl and Python are considered minor environments because they are not generally used for full-fledged application programming).



A Cocoa application being developed using Xcode. Interface Builder windows are also visible.

Cocoa applications are typically developed using the development tools provided by Apple, specifically Xcode (formerly Project Builder) and Interface Builder. However the Cocoa programming environment can be accessed using other tools, such as Perl, Python and PyObjC.

For end-users, Cocoa applications are considered to be those written using the Cocoa programming environment. Such applications usually have a distinctive feel, since the Cocoa programming environment automates many aspects of an application to comply with Apple's Human User Interface guidelines. As such, Cocoa applications are generally characterised by sleek, clean interfaces, and good performance.

Contents

Cocoa history

Cocoa is derived from the NeXTSTEP and OPENSTEP programming environments developed by NeXT in the late 1980s. Apple acquired NeXT in December 1996, and subsequently went to work on the Rhapsody operating system that was supposed to be the direct successor of OPENSTEP and use OPENSTEP technology proper. It was to have an emulation base for Mac OS applications, called *Blue Box*. The OPENSTEP base of libraries and

binary support was termed *Yellow Box*. Rhapsody evolved into Mac OS X, and the Yellow Box became Cocoa.

Much of the work that went into developing OPENSTEP was applied to the development of Mac OS X. Cocoa is the most visible part of that synergy. There are, however, some important fundamental differences. The most visible of which is that NeXTSTEP and OPENSTEP used Display PostScript for on-screen display of text and graphics, while Cocoa depends on Apple's Quartz (which uses PDF). Cocoa also has a level of internet support, including the NSURL and WebKit HTML classes, and others, while under OPENSTEP there was only rudimentary support for managed network connections through NSFileHandle classes and Berkeley sockets.

Prior to its current use, the "Cocoa" trademark was the name of an application that allowed children to create multimedia projects. It was originally known as KidSim, and is now licensed to a third party and marketed as Stagecast. The program was discontinued in one of the rationalizations that followed Steve Jobs' return to the company.

Memory management

One feature of the Cocoa environment that is certainly unusual, if not unique, is its facility for managing dynamically allocated memory. Cocoa's NSObject class, from which most classes, both vendor and user, are derived, implements a reference counting scheme for memory management. Objects derived from the NSObject root class respond to a `retain` and a `release` message, and keeps a retain count, which can be queried by sending a `retainCount` message. A newly allocated object, created with `alloc`, has a retain count of one. Sending that object a `retain` message increments the retain count, while sending it a `release` message decrements the retain count. When an object's retain count reaches zero, it is deallocated and its memory is freed. (Deallocation is to Objective C objects as destruction is to C++ objects. The `dealloc` method is functionally equivalent to a C++ *destructor*.)

In addition to manual reference counting, application programmers may choose to make use of *autorelease pools*. Sending an object an `autorelease` message registers a future release message with that thread's global autorelease pool. When the autorelease pool is deallocated, it sends the corresponding `release` message for every registered `autorelease`. Autorelease pools are generally deallocated and re-created at the end of the program's event loop, guaranteeing program flow has passed out of the block where that object was autoreleased. This means the application has predictable performance and memory collection is generally invisible to the user, whereas under most fully automated schemes the application will sometimes suddenly stop responding when the garbage collection system is started.

Cocoa gives the programmer the choice of whether to manually manage memory of objects or not. Opinions on this are divided. Some say that Cocoa's memory management is superior because it allows the programmer to have precise control over when his objects are deallocated, but does not burden him with the necessity of doing so for every object a program allocates, nor incurs the performance penalty that usually goes with automatic garbage collection. Others say that the entire scheme is unnecessary, and that Java-style automatic garbage collection is superior, because it removes the possibility of programmer error in memory management.

A combination of the two features is also possible. Modern garbage collectors often include features to be started and stopped mid-task, allowing the application to control how much time will be taken up whenever the system is called. Combining such a system with AppKit's "do it in the event loop" appears to offer a best-of-both-worlds solution. Such a system was successfully implemented under GNUStep, GNU's open source analog of OpenStep.

Main frameworks

Cocoa consists primarily of two Objective-C object libraries called *frameworks*. Frameworks are a construct

unique to the NeXTSTEP/OpenStep/Cocoa family of programming environments. They are functionally similar to shared libraries, a compiled object that can be dynamically loaded into a program's address space at run-time, but frameworks add associated resources, header files, and documentation. Cocoa also includes a powerful versioning system to prevent the sort of problems that occur under Microsoft Windows, DLL Hell.

- *Application Kit* or *AppKit* is directly descended from the original NeXTSTEP Application Kit. It contains code with which programs can create and interact with graphical user interfaces. NSWindow and NSButton are examples of AppKit classes.

- *Foundation Kit*, or more commonly simply **Foundation**, first appeared in OpenStep. Foundation is a generic object-oriented library providing string and value manipulation, containers and iteration, distributed computing, event handling, and other functions that are not directly tied to the graphical user interface. NSString, NSDictionary and NSURLHandle are examples of Foundation classes. The "NS" prefix, used for all framework objects, comes from Cocoa's NeXTSTEP/OpenStep heritage.

A key part of the Cocoa architecture is its comprehensive views model. This is organised along conventional lines for an application framework, but is based on the PDF drawing model provided by Quartz. This allows creation of custom drawing content using PostScript-like drawing commands, which also allow automatic printer support and so forth. Since the Cocoa framework manages all the clipping, scrolling, scaling and other chores of drawing graphics, the programmer is freed from implementing basic infrastructure and can concentrate only on the unique aspects of an application's content.

Model, View, Controller

The Smalltalk teams at Xerox PARC eventually settled on a design philosophy that led to easy development and high code reuse. Known as **Model-view-controller**, or **MVC**, the concept breaks an application into three sets of interacting object classes. Cocoa's design is a strict application of MVC principles.

In more recent versions of Cocoa, those shipping with OS X 10.3 and later, Apple has started to provide pre-rolled controller objects. Apple uses the term "bindings" to refer to these controllers, since they "bind" data in the model to controls and other elements in the user interface.

In Mac OS 10.4, Cocoa provides automatic support for the data model too, called Core data. By providing framework support for all three MVC layers, developers are freed from writing much boilerplate-type code, and can spend time on more worthwhile areas, such as the features that make their application unique.

Invocations and bindings

In most object oriented languages, calls to methods are represented physically by a pointer to the code in memory. This constrains the design of an application since specific "command handling" classes are required, usually organised according to the chain of command design pattern. While Cocoa retains this approach for the most part, the late binding possible with Objective-C opens up more flexible possibilities.

Under Objective-C calls can be represented instead by an **invocation**, essentially a string describing the call to be made. When a call is made, the invocation is sent into the ObjC runtime, matched against a list of possible methods, and then called. Since the invocation is text data, this allows it to be saved to a file, transmitted over a network, or manipulated in other ways. The "binding" between the call and the function is deferred until runtime, rather than needing to be resolved at compile time. There is a performance penalty for this, but this is small and outweighed by the advantages. The Cocoa GUI builder, **Interface Builder (IB)**, makes extensive use of this facility.

By a similar token, Cocoa provides a pervasive data manipulation technique called **Key-Value Coding** (KVC). This permits a piece of data or property of an object to be looked up or changed at runtime by name - the property name acts as a key to the value itself. In traditional languages, this late binding is not possible, and it leads to great design flexibility - an object's type does not need to be known, yet any property of that object can be discovered using KVC. In addition, by extending this system using something Cocoa called **Key-Value Observing** (KVO), automatic support for Undo/Redo is easily provided.

Rich objects

One of the most useful features of Cocoa are the powerful "base objects" the system supplies.

As an example, consider the Foundation class NSString, and the NSText system that manipulates it in the GUI. As well as providing storage for unicode character data, it provides many methods for doing complex manipulations of strings.

NSText and its related classes are the objects used to display and edit strings. The collection of objects involved permit the same code to implement anything from a simple single line text entry field to a complete multi-page, multi-column text layout schema, with full professional typography features such as kerning, ligatures, running text around arbitrary shapes, rotation, and of course full unicode support and anti-aliased glyph rendering. Paragraph layout can be controlled automatically or by the user, using a built-in "ruler" object that can be attached to any view. Spell checking is automatic, using a single universal dictionary used by all applications that uses the "squiggly underlining" introduced by Microsoft. Unlimited Undo/Redo support is built in. Using only the built-in features, one can write a text editor application in as few as 13 lines of code. With new controller objects this may fall to zero. This is in contrast to the TextEdit APIs found in the earlier MacOS.

When extensions are needed, Cocoa's use of Objective-C makes this a straightforward task. ObjC includes the concept of "categories" which allow code to be added onto existing classes at runtime. Adding functionality can be accomplished in a category without any changes to the original classes in the framework, or even access to its source. Under more common frameworks this same task would require the programmer to make a new subclass supporting the additional features, and then change all instances of the classes to this new class.

Implementations

The Cocoa frameworks are written in Objective-C, and hence Objective-C is presently the preferred language for the Cocoa applications. Java bindings for the Cocoa frameworks are also available, but do not yet appear to have seen much real-world use, and will no longer be updated after OS 10.5. In addition, the necessity of run-time binding means that many of Cocoa's key features are not available when using Java. AppleScript Studio, part of Apple's Xcode Tools makes it possible to write (less complex) Cocoa applications using AppleScript.

Third party bindings are also available for other languages:

- Python - PyObjC (http://pyobjc.sourceforge.net/)
- Ruby - RubyCocoa (http://rubycocoa.sourceforge.net/)
- Perl - CamelBones (http://camelbones.sourceforge.net/)
- C# - Cocoa# (http://www.cocoasharp.org/)

A more extensive list of implementations (http://www.fscript.org/links.htm) is available.

There is also an open source implementation of major parts of the Cocoa framework that allows cross-platform (including MS Windows) Cocoa application development. It is called GNUstep.

References

- Aaron Hillegass: *Cocoa Programming for Mac OS X*, Addison-Wesley, 2nd Edition 2004, Paperback, ISBN 0321213149.
- Stephen Kochan: *Programming in Objective-C*, Sams, 1st Edition 2003, Paperback, ISBN 0672325861.
- Michael Beam, James Duncan Davidson: *Cocoa in a Nutshell*, O'Reilly, 1st Edition 2003, Paperback, ISBN 0596004621.
- Erick Tejkowski: *Cocoa Programming for Dummies*, 1st Edition 2003, Paperback, ISBN 0764526138.
- Simson Garfinkel, Michael K. Mahoney: *Building Cocoa Applications : A Step by Step Guide*, O'Reilly, 1st Edition 2002, Paperback, ISBN 0596002351.
- James Duncan Davidson: *Learning Cocoa with Objective-C*, O'Reilly, 2nd Edition 2002, Paperback, ISBN 0596003013.
- Scott Anguish, Erik M. Buck, Donald A. Yacktman: *Cocoa Programming*, Sams, 1st Edition 2002, Paperback, ISBN 0672322307.
- Bill Cheeseman: *Cocoa Recipes for Mac OS X*, Peachpit Press, 1st Edition 2002, Paperback, ISBN 0201878011.
- Andrew Duncan: *Objective-C Pocket Reference*, O'Reilly, 1st Edition 2002, Paperback, ISBN 0596004230.
- Apple Computer Inc.: *Learning Cocoa*, O'Reilly, 1st Edition 2001, Paperback, ISBN 0596001606.

External links

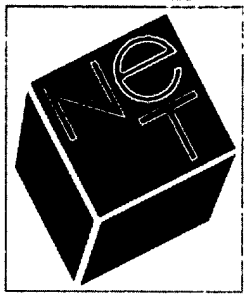
- Apple's Cocoa documentation (http://developer.apple.com/documentation/Cocoa/Cocoa.html)
- stepwise.com Cocoa Starting Point (http://www.stepwise.com/StartingPoint/Cocoa.html)
- PyObjC (http://pyobjc.sourceforge.net/)
- Cocoa Dev Central (http://www.cocoadevcentral.com/)
- Cocoa Development Wiki (http://www.cocoadev.com/)
- iDevApps (http://www.idevapps.com/)
- iDevGames (http://www.idevgames.com/)
- Project Cocoa (http://homepage.mac.com/redbird/cocoa/) — About the "old" Cocoa, also known as KidSim
- GNUstep (http://www.gnustep.org/) - cross-platform framework based on the OpenStep and major parts of Cocoa.

Categories: Mac OS X | Application programming interfaces

NeXT

From Wikipedia, the free encyclopedia.

NeXT was a computer company, known to the public for its series of futuristic computers, and to the programming world for its development platforms. NeXT merged with Apple Computer in 1997, in a buyout estimated at $400 million, and no longer does business as a separate entity.



The NeXT logo, designed by Paul Rand.

Contents

Prehistory

In 1985 Steve Jobs began to regret hiring John Sculley as the new CEO of Apple and started a brief power struggle to regain control of the company. The board stood behind Sculley, and Jobs was stripped of most of his duties and banished to an office at the back of a distant building on the Apple campus unofficially known as "Siberia". After a few months of being ignored, he left.

A few years later he found direction, when he decided that computers really were his strong suit, and started visiting various universities to look at where the industry was going. He concluded several technologies were going to be the next source of change:

- PostScript, which appeared to be on its way to becoming the standard graphics language
- Mach, which seemed to be re-writing the whole idea of the operating system
- object oriented programming (particularly using the Objective C language), a hot topic in the research world

Later that year he collected these ideas into a product concept that he thought would be the next big thing: an object oriented toolkit, aimed primarily at the academic market, using PostScript as the display technology.

Starting **NeXT Inc.** with an out-of-pocket investment of $7 million, he hired seven employees (mostly ex-Apple folk from the Apple Macintosh project) and started work with Adobe on what would eventually become Display PostScript.

NeXT Computer

Soon after NeXT, Inc. was formed Apple brought a lawsuit against the company. In an out of court settlement between the two parties, as of January 1986, NeXT was restricted to the workstation market.

By the middle of 1986 it was clear that no existing operating system (OS) was capable of hosting the toolkit, at least not on a personal computer level. Instead of making and selling a toolkit, the business plan changed to

making and selling complete machines running it on top of a Unix-like Mach-based OS. The latter would be created by a team led by Avie Tevanian, one of the Mach engineers at Carnegie-Mellon University who had since joined the company. The name of the company was changed to **NeXT Computer Inc.**.

By 1987 NeXT finished construction of a completely automated factory for their first product, the **NeXTcube**. Stories about Jobs' demands for the factory and the cube are now legend, including the re-painting of the factory several times in order to get just the right shade of grey, and the institution of a series of time consuming changes to the production line so that the cube's expensive magnesium case would have perfect right-angle edges.

Another example of what appears to be hubris can be seen in the selection of a drive mechanism. At the time most machines shipped with hard drives of 20 or 40MB, onto which software (including the OS) was loaded using floppy disks. Even in the late 1980s this was starting to be a real problem, as the user needed to swap huge stacks of floppies to load the ever-growing applications.

This was even more of a problem for NeXT. Even the hard drive didn't solve their problem because the OS was several tens of MB, and the stack of floppies needed to load it would be bigger than the machine. Larger hard drives were available but they were terribly expensive. At the time a usably-large 640MB drive cost $4995.

So instead NeXT would try to do one better, replacing both the hard drive and floppy with a single removable medium. This was in the form of a 256MB magneto-optical device made by Canon. Magneto-optical drives were just coming to market, and the one in the NeXT cube was the first to ship. This was a very risky move considering the equipment didn't even exist during the design stages, and many have claimed it was used primarily due to Jobs' disdain for the floppy.

The Cube was based on a 25MHz Motorola 68030 CPU which had recently come to market, making it competitive with the workstation vendors like Sun Microsystems in terms of performance. There had been some discussion of using the Motorola 88000 RISC chip, but it was considered too risky as they weren't available in quantity at the time.

The 68030 was supported by the 68882 FPU for faster math, the 56001 DSP for multi-media work, and two custom-designed 6-channel DMA channel controllers which allowed much of the I/O to be offloaded from the main processor to boost the speed of common tasks.

The Cube fit into an odd spot in the computer market. It wasn't as fast as the latest generation of Unix workstations becoming available at that time, but cost about half as much. Comparing the Cube with more common Intel based machines was more difficult. The machine shipped with a huge 8MB of RAM (at a time when 4MB cost $1495), the 256MB MO drive, Ethernet, NuBus and a large "megapixel" (1120 x 832 pixel) greyscale display. Meanwhile the typical PC still used the 8088, the 8086 or 286 CPU, had either a 320x200 4-color or 640x480 black and white display, typically had no networking, and may or may not have a hard drive. Could those machines even compare with the NeXT at all?

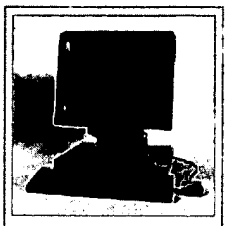
Prototype Cubes were shown to standing ovations in October 1988, and a slew of magazines reviewed the system - all concentrating on the hardware. By 1989 the machines were in beta form, and they started selling limited numbers to universities with a 0.9 version of the OS installed. (When asked if he was upset that the computer's debut was delayed by several months, Jobs responded, "Late? This computer is five years ahead of its time!")

The machines weren't ready for "real" sales until 1990, when they went on the market for $9999. At the time Jobs was concerned that the market was quickly stratifying and the window of opportunity to introduce any new platform was rapidly closing. Just after their release he noted that "this will either be the last machine to make it, or the first to fail".

When it was discovered that the MO drive led to very serious performance problems in real-world use (as well as

costing about $100 per disk), NeXT as a whole gained a reputation for failure that would never rub off. Basically the drive itself, while faster than a floppy, was simply not fast enough to run a Unix based OS as its primary medium. But more annoyingly, with the OS loaded onto the disk, simply copying a file from one disk to another was almost impossible, as removing the disk removed the OS along with it. And since most other machines didn't have networking, and instead used floppies for moving data files around (the so-called sneakernet) it was equally difficult to move files to and from the machine.

This problem was rectified by 1991, when a new series of machines with floppy disks and hard drives shipped. A new line then introduced the newer and much faster 68040. The same parts were later put in a new "pizza box" case, creating the **NeXTstation**, which sold at a lower price point and became fairly popular. In the NeXT community, these machines were commonly referred to as the "NeXT slab".


NeXTstation

With all of the attention focused on the hardware, the true gem of the system, NeXTSTEP, was lost in the hype. Nevertheless, NeXT staff frequently wrote articles in major programming magazines such as Dr. Dobb's, showing how some recent article's 3+ pages of code was implemented under NeXTSTEP in perhaps 10 lines.

A number of programs started shipping for the system, including the acclaimed Lotus Improv spreadsheet, and WorldWideWeb the world's first web browser. The system also shipped with a number of "smaller" applications built in that would actually improve the environment considerably without being obvious, things like the Merriam-Webster Collegiate Dictionary, Oxford Quotations, the complete works of William Shakespeare, and the *Digital Librarian* search engine to access them all.

In all, some 50,000 NeXT machines were sold. This was a tiny segment of the market, and proved Jobs' own words prophetic. Although the lack of success by other new desktop platforms (such as the BeBox) suggests that the age of unique hardware designs was past, it is an open question as to whether the systems would have been more successful had they avoided the performance and price problems by including a hard drive in the first machines, and had found a more cost-effective RAM setup.

One of the machines was used in 1991 by Tim Berners Lee when he created the first web browser and web server. This was the beginning of the world wide web as the world knows it today.

NeXT Software

By 1992 work had already started on a port of the NeXTSTEP operating system to the Intel platform. At the same time work began on replacing the 68000 series CPU's with the new PowerPC, which was starting up as a joint program between Apple, IBM and Motorola.

By late 1993 the Intel port was complete, and was released in the form of NeXTSTEP 3.1 (also referred to as NEXTSTEP 486). Work on the PowerPC machines was stopped along with all hardware production. The company renamed once again, this time to **NeXT Software Inc.**

NeXTSTEP 3.x was later ported to PA-RISC and SPARC based platforms, for a total of four versions:

- NeXTSTEP/NeXT (for NeXT's 68k "black boxes")
- NeXTSTEP/Intel
- NeXTSTEP/PA-RISC
- NeXTSTEP/SPARC

None of the non-NeXT versions appear to have seen much use, overall. However, it did gain popularity at

institutions such as the Central Intelligence Agency, First Chicago NBD, Swiss Bank Corporation, and other organizations, for the popularity of the programming model. At the time, the performance of the Intel platforms was quite limited (although not for long), and running it on the other two systems meant replacing their "native" OSes outright. One of the primary reasons for buying one of these platforms was to use specialized software that ran only on the their operating system/cpu combination (as opposed to today, where the most common use is as a server), and running NeXTSTEP meant giving that up.

At this point NeXT's attention turned away from supplying a complete OS, and along with Sun Microsystems they started an effort that would lead to OPENSTEP. This was basically NeXTSTEP without the Mach-based Unix underneath it, using some other OS instead.

The company had now come full circle. Originally intending to sell a toolkit running on top of other OSes, they had ventured into hardware, failed, and returned to selling a toolkit running on top of other OSes. Although OPENSTEP had an enthusiastic audience of developers using it for enterprise software and the like, it never attracted really large numbers of paying customers, and lack of revenue growth was a perennial problem.

New products based on OPENSTEP continued to ship, including OPENSTEP ENTERPRISE a version which ran all OPENSTEP applications and frameworks on Windows NT. The company also launched WebObjects which was one of the first Application Server platforms for building dynamic enterprise level projects. This technology is still in use in a few online stores including Apple's breakthrough iTunes Music Store.

End of NeXT

In 1996 Apple Computer purchased NeXT Software in order to use NeXTSTEP to replace the now outdated Mac OS.

Steve Jobs returned to Apple as a consultant, then as interim CEO (or "iCEO", echoing the name of Apple's new iMac consumer hardware), and finally as CEO. With him, he brought most of the NeXT executives, who replaced their Apple counterparts. Industry commentators summarized this by referring to the acquisition as "NeXT getting paid to buy Apple".

Over the next four years the NeXTSTEP operating system was ported to the Apple Macintosh PowerPC architecture, and the Intel version and the OpenStep Enterprise toolkit for Windows were kept in sync. The operating systems were codenamed Rhapsody, while the toolkit for development on all platforms gained the moniker *Yellow Box*. Apple added much of their facilities and tools to Rhapsody, including QuickTime and ColorSync. For backwards compatibility Apple added the *Blue Box* to the Mac version of Rhapsody to allow existing Mac applications to be run in a self-contained environment.

After two beta releases Rhapsody disappeared, and with it the Intel and Windows versions. The operating system was released as Mac OS X and the OpenStep toolkit was named Cocoa. At the insistence of existing Mac developers, Apple included a cut-down version of the original Macintosh toolbox that allowed existing Mac apps, with some modification, integrated access to the environment without the constraints of Blue Box. This was named Carbon.

However, NeXTSTEP's cross-platform capabilities were kept intact within Mac OS X. Every version was compiled onto both the PowerPC and Intel x86 architectures, even though only the PowerPC version was released. In 2005, Apple announced that starting in 2006, Macintoshes would be based on Intel CPUs instead of PowerPCs, returning the NeXT software back to the platform it was first ported to in 1993.

See also

- GNUstep, a Free implementation of OpenStep

External links

- The Chronology of Workstation Computers (http://www.islandnet.com/~kpolsson/workstat/)
- From NeXT To X (http://mlagazine.com/modules.php?op=modload&name=News&file=article&sid=17)

Retrieved from "http://en.wikipedia.org/wiki/NeXT"

Categories: Software companies | Computer hardware companies | Computer workstations | NeXT | Computer companies of the United States | Defunct computer companies of the United States

Chronology of Workstation Computers

This document is an attempt to bring various published sources together to present a timeline about Workstation Computers. I do not have a good definition of what a 'Workstation Computer' is (or is not), nor do I believe I will find one that everyone can agree on. Indeed, it is likely that any such definition has changed over time, especially if such a definition was based on absolute processing power.

The characteristics of a typical workstation help determine if a particular system is or is not a workstation. The main characteristics I have found are as follows. A 'Workstation Computer' typically:

- runs Unix, or a similar operating system,
- has leading-edge processor power,
- has a large graphics display,
- has more memory and disk space than other desktop computers,
- is used for scientific research, CAD, real-time simulations, animation, and, as a result of the above,
- costs more than most high-end desktop computers.

This timeline is an off-shoot of my research on a Chronology of Personal Computers. I am adding info to this timeline as I encounter it, without looking too hard for large quantities of information at this time.

This complete timeline is provided on the Web for free. If you wish to contribute money to help my keeping this timeline available and updated, I would very much appreciate it.

References are numbered in [brackets], which are listed in this file. A number after the dot gives the page in the source.

Last updated: 2005 April 29.

1968-1986 | 1987-1990 | 1991-1992 | 1993 | 1994 | 1995-end

1968

(month unknown)

- In California, IBM scientist John Cocke and others complete a prototype scientific computer called the ACS. It incorporates some RISC concepts, but the project is later cancelled due to the instruction set not being compatible with IBM's System/360 computers. [95.40]

1969

(month unknown)

- At Bell Laboratories in Murray Hill, New Jersey, Ken Thompson and Dennis Ritchie write a primitive operating system in assembly language on the PDP-7. This becomes the Unix operating system. [110.148] [127.168] [50.116] [51.10] [67.24] [156.ss8] [202.74]

1970

(month unknown)

- Brian Kernighan suggests naming the operating system written by Ken Thompson and Dennis Ritchie's "Unix", as a pun on Multics, the operating system that it was to replace. [156.ss8]

1972

(month unknown)

- Xerox decides to build a personal computer to be used for research. Project "Alto" begins. [52.58] [54.267] [109.85]

November

- Researchers at PARC begin work on a prototype Alto personal computer. [109.93]

1973

March

- The first prototype Alto workstation computer is turned on at Xerox' Palo Alto Research Center. Its first screen display is a bitmapped image of the Sesame Street character Cookie Monster. [40.59] [109.14] [109.93] (completed in 1974 [54.267])

April

- The first operational Alto computer is completed at Xerox PARC. [108.95,167]

1974

(month unknown)

- Xerox releases the Alto computer. [54.xv]
- IBM scientist John Cocke completes a prototype high-reliability, low-maintenance computer called the ServiceFree. It incorporates a RISC architecture, achieving at least 80 MIPS, 50 times faster than IBM's fastest mainframe at the time. However, the project is later cancelled due to the massive "Future Systems" project consuming much of IBM's resources. [95.40]

1975

June

- At Xerox, John Ellenby proposes they build the Alto II personal computer, a modified Alto, making it easier to produce, more reliable, and more easily maintained. His request is approved. [109.205]

1976

(month unknown)

- Xerox management rejects two proposals to market the Alto computer. [109.174]

1977

(month unknown)

- Xerox renames its Janus workstation project to Star. [109.231]

1978

(month unknown)

- At the University of California at Berkeley, programmers add virtual memory control to the Unix operating system. [174.348]
- IBM scientist John Cocke produces the 801 computer, a RISC prototype named after the laboratory building it was built in. This minicomputer is often considered the invention of RISC processing. [95.42]

1979

September

- Motorola introduces the 68000 16-bit microprocessor. It uses 68,000 transistors, giving it its name. [38.75] [71.136] (1980 [20])

1980

(month unknown)

- IBM's Austin Laboratory in Burlington, Vermont, creates a microprocessor called ROMP (Research/Office Products MicroProcessor) based on Jon Cocke's 801 RISC design. This is the first completed RISC microprocessor. Speed is about 5-10 times faster than any other chip on the market. [95.42]
- Apollo introduces a line of workstations using the Motorola 68000 processor. [40.90]
- The term RISC (reduced instruction set computer) is coined by Professor David Patterson of the University of California in Berkeley. He designs a microprocessor called RISC I. [23] [95.40]

1981

(month unknown)

- James Clark invents the Geometry Engine 3-D software. [221.61]
- College professor James Clark found Silicon Graphics, Incorporated. [28] (1982 [79.9])

1982

February

- Scott McNealy, Bill Joy, Andreas Bechtolsheim, and Vinod Khosla found Sun Microsystems. "SUN" originally stood for Stanford University Network. [47] [110.149,152] [217.163]

May

- Sun Microsystems begins shipping the Sun 1 workstation computer. [110.152]

November

- At the COMDEX show, Victory Computer Systems announces the Victory Factor series of computers, using the Motorola 68000 processor and the Unisoft Uniplus System 3 Unix operating system. [128.268]

(month unknown)

- Toshiba introduces the Tosbac UX-300. It features a Toshiba 88000 processor, 512 kB RAM, 1 MB 8-inch floppy drive, 10 MB hard drive, and runs UNIX, for US$9300. [74.113]

1983

January

- AT&T announces UNIX System V. [76.133]

(month unknown)

- IBM's Austin Laboratory begins project Olympiad, to develop a scientific workstation based on the ROMP microprocessor. [95.45]
- Fortune Systems introduces the Fortune 32:16 computer system. It features a 6 MHz Motorola 68000 CPU, 256 kB RAM, and runs Unix v7. Price is US$5000-11,000. [128.84]

November

- Silicon Graphics introduces its first product, IRIS 1000 terminal. [221.61]

(month unknown)

- Commodore Business Machines announces it will include the Coherent Unix-like operating system on a new series of Z8000-based computers called the Next Generation. [135.7]

1984

February 14

- Scott McNealy is appointed president and chief operating officer of Sun Microsystems. [110.153] [218.D2]

April

- Silicon Graphics begins shipping its first 3-D graphics workstations. [28]

June

- Motorola introduces the 16 MHz 68020 processor, a 32-bit version of the 68000, in CMOS, with on-board cache. [1] [140] (1986 [20])

(month unknown)

- MIPS Computer Systems is founded, and begins developing its RISC architecture. [29]
- Sun Microsystems co-founder Vinod Khosla resigns. [110.153]
- Silicon Graphics introduces its first workstation, IRIS 1400. [221.61]

1985

(month unknown)

- Sun Microsystems begins work on its SPARC processor. [29]
- AT&T grants a UNIX license to IBM. [219.1]

September

- Steve Jobs and five former senior managers of Apple Computer Inc. found NeXT Incorporated. [33.66] [46] [42.289] [65.213]

1986

January

- Sun Microsystems first sells shares to the public. [110.219]
- IBM announces the IBM RT Personal Computer, using RISC-based technology from IBM's "801" project of the mid-70s. It is one of the first commercially-available 32-bit RISC-based computers. The base configuration has 1 MB RAM, a 1.2 MB floppy, and 40 MB hard drive, for US$11,700. With performance of only 2 MIPS, it is doomed from the beginning. [6] [19] [41.114] [61.129]
- NeXT and Apple Computer reach an out-of-court settlement on Apple Computer's lawsuit against NeXT. [110.99]

March

- Silicon Graphics decides to switch from the Motorola 68000 processor line to MIPS Technologies' RISC processors. [29]

(month unknown)

- MIPS Technologies unveils the 8 MHz R2000 32-bit CPU. With 110,000 transistors, it achieves a speed rating of 5 MIPS. [38.75] (1985 [42.124])
- MIPS Technologies begins volume shipments of the 8 MHz R2000 processor. [29]

June

- Systems incorporating MIPS Technologies' R2000 processor begin shipping. [89.13] [187]

(month unknown)

- IBM begins work on a new line of Unix-based workstations. (*They will become the IBM RS/6000 series.*) [26]
- Motorola begins work on the 88000 processor. [29]

September

- Steve Jobs decides to use erasable optical disk drives for the first NeXT computer. [33.66]

November

- The TV show, "The Entrepreneurs" airs nationally on PBS in the US. One segment shows Steve Jobs and his NeXT employees discussing business at a company retreat. [110.97]

End of 1968-1986. Next: 1987-1990.

1968-1986	1987-1990	1991-1992	1993	1994	1995-end

This complete timeline is provided on the Web for free. If you wish to contribute money to help my keeping this timeline available and updated, I would very much appreciate it.

A list of **references** to all source material is available.

Also check my list of other timelines.

Chronology of Workstation Computers

internet e-mail: kpolsson@islandnet.com

URL: http://www.islandnet.com/~kpolsson/workstat/

This complete timeline is provided on the Web for free. If you wish to contribute money to help my keeping this timeline available and updated, I would very much appreciate it.

References are numbered in [brackets], which are listed in this file. A number after the dot gives the page in the source.

Last updated: 2005 April 29.

1968-1986 | **1987-1990** | 1991-1992 | 1993 | 1994 | 1995-end

1987

July 8
- Sun Microsystems introduces its first SPARC-based system, the Sun-4/260, with 10 MIPS performance. Sun announces it is offering licenses for its SPARC microprocessor architecture. [29] [34.80] [38.75] [216.D3] (October [36.56]) (late 1986 [95.37])

December 31
- Market share of Technical/Engineering workstations: Sun Microsystems 21.0%, DEC 23.1%, Apollo 19.3%, HP 14.6%, IBM 6.8%. [57.222]

1988

January
- Sun Microsystems and AT&T announce the intention to set a standard for UNIX. [220.369]

April
- Motorola unveils the 88000 processor. [29]

May
- A consortium of seven companies, including IBM, announce the Open Software Foundation to develop UNIX standard. [220.369]

(month unknown)
- Sun Microsystems ships its 100,000th workstation computer. [197.9]
- NeXT begins negotiations with Businessland for a possible deal to sell NeXT computers outside of the education market. [110.201]

September
- SPEC is formed, with the aim of producing a Unix system benchmark based on a standard set of real-life applications programs. [25]

(month unknown)
- An engineering task force at Digital Equipment begins project Alpha, to develop a new processor architecture to succeed the VAX. [69.61] [116.141] (begins in mid-1989 [99.7])

October

- Steve Jobs begins persuading Lotus Development to develop the Improv spreadsheet program for the NeXT computer. [110.266]
- Steve Jobs of NeXT Inc. unveils the first NeXT computer, at the Davies Symphony Hall in San Francisco. For US$6500, it features: 25 MHz Motorola 68030 processor and 68882 math coprocessor, 8 MB RAM, 17-inch monochrome monitor, 256 MB read/write magneto-optical drive, and object-oriented NeXTSTEP operating system. It is dubbed the "Cube" because its system box measures 1 foot on all sides. [21] [33.65] [39.76] [40.7] [42.289] [64] [65.280] [110.14,162,166] [214] (August [2])

(month unknown)

- Quote by Bill Gates, of Microsoft, on Steve Jobs' introduction of the NeXT computer: "He put a microprocessor in a box. So what?". [110.14]
- Quote by Bill Gates, of Microsoft, when asked if he would develop software for the NeXT computer: "Develop for it? I'll piss on it.". [110.14]

December 31

- Shipments of Unix-based systems in Italy for the year: 30,000, worth US$548 million. [166.18]

1989

January

- Digital Equipment introduces its first RISC-based workstation, the DECstation 3100, using the 16.7 MHz R2000 MIPS Technologies processor. [24] [29] [173.201]

February

- Quote from Sun Microsystem's Scott McNealy when asked what he thought of the NeXT Cube: "... it's the wrong operating system, the wrong processor, and the wrong price.". [110.212]

March

- NeXT announces a deal with Businessland to sell NeXT computers outside of the education market. [110.201,208]

May

- Hewlett-Packard buys workstation maker Apollo Computer for US$476 million. [27]
- Solbourne Computers Incorporated is the first to announce a line of SPARC-based Sun-compatible computers. [34.80] (1988 [35.81])

June

- Canon agrees to buy 16.67% of NeXT for US$100 million. [110.218]

(month unknown)

- Digital Equipment introduces the DECstation 2100 computer. It features 12.5 MHz R2000 processor, 8 MB RAM, 15 inch monochrome 1024x864 monitor, Ethernet, mouse, keyboard, Ultrix Workstation Software operating system, SoftPC DOS emulator. Price is US$7950. [166.49] [173.201]
- Sun Microsystems announces the 12.5 MIPS 20 MHz SPARCstation 1 for a base price of US$9000. [4] [38.58] [145]
- SPARC International is formed. [34.80]
- Data General unveils its Aviion workstation line, based on the Motorola 88000. [29]

July

- In Tokyo, Canon introduces the NeXT Computer. Price is about US$14,000. An optional 660 MB hard drive adds US$14,000. [166.26]

(month unknown)

- Absoft introduces a FORTRAN 77 compiler for NeXT systems. Price is US$1000. [173.66]
- To date, there are about 4 to 5 million Unix users worldwide. [174.364]
- Cypress Semiconductor introduces the 40 MHz 7C601 RISC processor, based on Sun Microsystem's SPARC design. Performance is about 29 MIPS. Price is US$895 in 100 unit

quantities. [173.18]
- Hewlett-Packard's Apollo division introduces the Apollo 2500 workstation. It features 20 MHz Motorola 68030 processor, 20 MHz 68882 math coprocessor, 4 MB RAM, 15 inch 1024x768 monochrome monitor, keyboard, mouse, Unix System V release 3. Price is US$3990. Price with 200 MB SCSI hard drive and Domain/OS operating system is about US$3990. [164.94] [163.49] [173.8] [201.30]

September
- Hewlett-Packard announces a US$3990 UNIX workstation based on the Motorola 68030. [27]
- NeXT ships the first NeXT Computer systems. [42.289] [46] [164.145]
- NeXT releases NeXTSTEP v1.0. [42.289] [46] [164.145]

October
- SPEC releases version 1.0 of its SPEC Benchmark Suite for Unix systems. [25] [145.6] [191]
- IBM signs a deal with NeXT to license the NeXTSTEP operating system, for US$10 million. [33.65] (1988 [96.310])

(month unknown)
- Motorola begins large volume shipments of the 88100 processor. [191]
- In New York, the Unix Expo is held. [164.30]
- Sun Microsystems publishes its SBus i/o bus as an open specification for its workstation computers. [165.283]

December
- IBM demonstrates its new line of RISC System/6000 workstations. [3]

December 31
- Shipments of Sun Microsystems SPARC workstations for the year: 45,000. [34.80]

1990

January
- Sun Microsystems signs an agreement to transfer the SPARC trademark to SPARC International. [34.80]
- NeXT decides to redesign the NeXT computer, targeting a cheaper computer to be available by the fall. [110.253]

February
- NeXT co-founder Dan'l Lewin resigns from NeXT. He is the first of the original five co-founders to resign. [110.255]

February 15
- IBM unveils its new RISC-based workstation line, the RS/6000. Development work had been done under code name "America" for the RISC chip research, and "RIOS" for systems using the America technology. The architecture of the systems is given the name POWER, standing for Performance Optimization With Enhanced RISC. [41.116] [110.282] [209.86]

May
- Toshiba unveils the first SPARC laptop, the SPARC LT. [34.80]

(month unknown)
- Lotus Development introduces Improv spreadsheet program for the NeXT Computer. Price is US$695. [176.147]

September
- Quote by James Clark, chairman of Silicon Graphics, on NeXT: "They're dead meat.". [212.52]

September 18
- At the Davies Symphony Hall in San Francisco, NeXT unveils three new NeXT computers, including the NeXTstation using a Motorola 68040 processor. [33.66] [42.289] [110.261] [212.50] (October [33.65])

October

- At the Microprocessor Forum, Motorola announces a new line of single-chip RISC processors, the first of which is to be the 88110. [41.81] [150] [182]
- IBM demonstrates the NeXTSTEP operating system running on an IBM RS/6000 workstation. [110.283]

November

- LSI Logic announces the availability of SparcKIT, a SPARC chipset at speeds of 20 MHz and 25 MHz. [34.80]

December 31

- Market share of workstation computers: Sun Microsystems, 29.1%; Hewlett-Packard, 22.7%; DEC, 17.7%. [31.13]
- Shipments of Sun Microsystems workstations for the year: 130,000. [34.80]
- Shipments of IBM RISC System/6000 computers for the year: 25,000. [32.13]

End of 1987-1990. Next: 1991-1992.

1968-1986 | **1987-1990** | 1991-1992 | 1993 | 1994 | 1995-end

This complete timeline is provided on the Web for free. If you wish to contribute money to help my keeping this timeline available and updated, I would very much appreciate it.

A list of **references** to all source material is available.

Also check my list of other timelines.

Last updated: 2005 April 29.

URL=http://www.islandnet.com/~kpolsson/workstat/
Link to Ken P's home page.

Chronology of Workstation Computers

This complete timeline is provided on the Web for free. If you wish to contribute money to help my keeping this timeline available and updated, I would very much appreciate it.

References are numbered in [brackets], which are listed in this file. A number after the dot gives the page in the source.

Last updated: 2005 April 29.

1968-1986 | 1987-1990 | **1991-1992** | 1993 | 1994 | 1995-end

1991

January
- Sun Microsystems begins shipping the SPARCstation 2. [35.81]
- RDI announces the availability of Macintosh emulation software for SPARC systems. [34.80]

February
- Sun Microsystems creates SunSoft, a system software subsidiary. [172.198]
- MIPS Technologies unveils the R4000 RISC processor architecture. [31.13]

March
- NeXT begins shipping of its low-end NeXTstation color computers for US$8000. [33.66] [110.263] (April [42.289])

May
- NeXT begins shipping of its high-end NeXTstation color computers for US$14000. [110.263]

June
- Ross Perot resigns from the NeXT board of directors. [110.292]

July
- NeXT completes the Japanese version of the NeXTSTEP operating system. [110.280]
- Sun Microsystems introduces the SPARCstation ELC, and the SPARCstation IPX. [34.80]

August
- Silicon Graphics announces the Indigo computer. [138.81]

(month unknown)
- Steve Jobs agrees with his NeXT company advisors to port the NeXTSTEP operating system to the Intel 80x86 architecture. [110.318]

September
- MIPS Technologies begins shipping samples of the R4000 processor. [98.9]
- Silicon Graphics begins licensing the OpenGL graphics library. [130.108]
- SunSoft announces the Solaris 2.0 operating system for 80x86 and SPARC architectures. [172.198]

October
- Sun Microsystems begins licensing the new chipset used in the SPARCstation 2. [35.81]
- MIPS Technologies officially introduces the 100 MHz R4000, its 64-bit RISC processor. [8] [48]

November

- MIPS Computer Systems announces its ARC System licensing program, consisting of a complete ARC System 100 system design, supporting ROM code, operating system drivers, and ASICs. [112.1]
- Intel decides against licensing Digital Equipment's technology in the Alpha architecture. [97.24]

December

- Standards Performance Evaluation Corp (SPEC) names its integer and floating point benchmark metrics for Unix systems as SPECint and SPECfp, respectively. [145.6]

December 31

- Shipments of Unix operating system for the year: 1.2 million. [175.134]

1992

January

- Standards Performance Evaluation Corp (SPEC) announces availability of SPECint92 and SPECfp92 benchmark suites for Unix systems. Cost of the suite is US$900. [121.78] [113] [145.6] [191]
- At NeXTWORLD Expo in San Francisco, NeXT announces that a version of the NeXTSTEP operating system will be made for Intel PCs. [42.289] [110.319]
- Standards Performance Evaluation Corp (SPEC) announces renaming of its Unix system benchmark metrics to include the "89" suffix, ie. SPECint89 and SPECfp89. [145.6]

(month unknown)

- Hewlett-Packard introduces the Series 9000 Model 710 computer. It features 50 MHz PA-RISC processor, Ethernet, serial, parallel, SCSI-2, 16 MB RAM, 19-inch 8-bit grayscale monitor, for US$9490. [161.36]
- Hewlett-Packard introduces the Series 9000 Model 705 computer. It features 35 MHz PA-RISC processor, Ethernet, serial, parallel, SCSI-2, 16 MB RAM, 19-inch 8-bit grayscale monitor, for US$4990. [161.36]

February

- Ross Technology publicly previews its Pinnacle-1 SPARC processor. [111]

February 25

- Digital Equipment unveils the 64-bit Alpha processor architecture, with speed estimates of 150 million instructions per second. [210.38] [213.63]

(month unknown)

- Tadpole Technology introduces the Sparcbook portable computer. It features 8-32 MB RAM, 85-240 MB hard drive, floppy drive, gray-scale or color 640x480 monitor, 25 MHz CY601 integer processor, 25 MHz CY604 floating point processor, Ethernet port, internal 9600/2400 fax/modem, Solaris operating system, Open Windows 3.0, SoftPC to run MS-DOS and MS-Windows applications. Size is 12 x 8.5 x 2 inches, weight is 7 pounds 1 ounce, price is US$4950-14850. [165.40]

March

- MIPS Technologies ships the 100 MHz R4000 processor. [90.134] [150]
- Silicon Graphics announces it is acquiring MIPS Computer Systems. [111.1]

(month unknown)

- MIPS Computer Systems announces the Magnum 4000 and Millennium 4000 OEM systems, based on the ARC System 100 design. Both use a 50/100 MHz R4000 processor. [112.1]
- Silicon Graphics ships its 100,000th workstation. [197.9]

May

- Ross Technologies announces the hyperSPARC processor. [180] [182]
- IBM announces the RS/6000 Model 970 computer. It features 50 MHz processor with 32 kB

instruction cache and 64 kB data cache, 64 MB RAM, 2.7 GB hard disk. Price is US$97,822. Performance specifications are 47.1 SPECint92, 93.6 SPECfp92, 49.3 SPECint89, 160.9 SPECfp89. [113.5]
- Sun Microsystems announces the SPARCstation 10 family, using the Sun / TI Viking SuperSPARC processor. Original name for the family was SPARCstation 3. [113.11] [116.20] [180]
- Sun Microsystems announces the SPARCstation 10 model 30. Features include 36 MHz SuperSPARC processor, 32 MB RAM, 424 MB hard drive Price is US$18495. [113.11]
- Sun Microsystems announces the SPARCstation 10 model 41. Features include 40 MHz SuperSPARC processor, 1 MB second-level cache, 32 MB RAM, 424 MB hard drive Price is US$24995. [113.11]
- Sun Microsystems announces the SPARCstation 10 model 52. Features include two 45 MHz SuperSPARC processors, each with 1 MB second-level cache, 64 MB RAM, 1 GB hard drive cache Price is US$39995. [113.11]
- Sun Microsystems announces the SPARCstation 10 model 54. Features include four 45 MHz SuperSPARC processors, each with 1 MB second-level cache, 64 MB RAM, 1 GB hard drive cache Price is US$57995. [113.11]

(month unknown)
- SPARC International introduces version 9 of the SPARC architecture. It features 64 bit address space, and instruction set extensions for superscalar implementations. [175.26]
- Unix Systems Laboratories announces Unix System V release 4.2. [175.38]
- Silicon Graphics and the OpenGL Architecture Review Board officially release OpenGL 1.0 Specification. [130.13]
- Bud Tribble, co-founder of NeXT, resigns from NeXT, and goes to work for Sun Microsystems. [110.330]

September
- NeXT ships NeXTSTEP v3.0. [42.289]

(month unknown)
- IBM announces the RS/6000 Model 580 computer. It incorporates a 62.5 MHz POWER chip set. Performance is 59 SPECint92, 125 SPECfp92, and 126 SPECmark89. Base price is US$62,500. [146]

October 14
- At the Microprocessor Forum, Texas Instruments and Sun Microsystems formally unveil the 50 MHz microSPARC processor. The processor includes integer and floating-point units, and 4 kB instruction and 2 kB data caches. It incorporates 800,000 transistors, using a 0.8-micron CMOS process. Development names during development were Tsunami and TMS390S10. Performance is about 40 MIPS. [90.134] [147.1] [107.36] [180]

October
- Hewlett-Packard announces the PA-7100LC processor. [182]

November 16
- Digital Equipment demonstrates systems with 125 MHz 21064 Alpha processors running a variety of Windows NT applications. [149]

November
- Digital Equipment unveils the 150 MHz DECchip 21064 microprocessor, implementing the Alpha AXP 64-bit architecture. Development name was EV-4. [7] [37.15] [69.61] [87.64] [90.134] [148.1] [187]
- Digital Equipment introduces the DEC 3000 Model 500 AXP, featuring 150 MHz Alpha 21064 processor, 512 kB cache, 32 MB RAM, 1 GB hard drive, six Turbochannel expansion slots, CD-ROM drive, and 19-inch color monitor. Price is US$38.995. Performance is 121.5 Specmark. [137.22] [148.1]
- Sun Microsystems introduces the SPARCclassic, featuring a 50 MHz microSPARC processor, 16

MB RAM, 207 MB hard disk, two SBus expansion slots, Solaris 2.1, and 15-inch color monitor. Price is about US$4500. Performance specs are 59.1 MIPS, 26.4 SpecInt92, 21.0 Specfp92. [37.15] [137.21] [148]

- Sun Microsystems introduces the SparcStation LX, featuring 50 MHz microSparc processor, 16 MB RAM, 424 MB hard disk, Solaris 2.1, GXplus video accelerator, CD-quality audio, built-in ISDN, and 16-inch color monitor. Price is US$7995. Performance specs are 59.1 MIPS, 26.4 SpecInt92, 21.0 Specfp92. [37.15] [137.21]
- Sun Microsystems announces SPARCcenter 2000 multiprocessor server. Up to 20 SuperSPARC processors can be used. Price with two processors is US$95,000. [37.15] [148]
- MIPS Technologies (division of Silicon Graphics) announces the R4400 microprocessor, previously called the R4000A. It is an R4000 processor with double on-chip cache, implemented in 0.6-micron CMOS. Clock rates of up to 150 MHz (75 MHz internal) will be available. The chip incorporates 2.2 million transistors. [148.1] [180]
- Hewlett-Packard announces the HP 9000 Model 735 desktop workstation. It features a 99 MHz PA-RISC 7100 processor, 32 MB RAM, 525 MB SCSI hard drive, and 19-inch color monitor. Rated speeds are 147 Specmarks and 40.8 MFLOPS. Price is US$37,395. [37.15] [139.18]
- Hewlett-Packard announces the HP 9000 Model 755 deskside workstation. It features a 99 MHz PA-RISC 7100 processor, 64 MB RAM, 2 GB hard drive, and 19-inch color monitor. Rated speeds are 147 Specmarks and 40.8 MFLOPS. Price is US$58,995. [37.15] [139.18]
- Hewlett-Packard announces the HP 9000 Model 715/33 workstation. It features 33 MHz PA-RISC processor, grayscale monitor, and one expansion slot. Price is US$4,995. [37.15] [139.18]
- Hewlett-Packard announces the HP 9000 Model 715/50 workstation. It features 50 MHz PA-RISC processor, grayscale monitor, 512 MB hard drive, and one expansion slot. Price is US$11,895. [139.18]
- Hewlett-Packard announces the HP 9000 Model 725/50 workstation. It features 50 MHz PA-RISC processor, 4-slots, grayscale monitor, 512 MB hard drive, 16 MB RAM, for US$17,895. [37.15] [139.18]

December

- Novell buys UNIX Systems Laboratories from AT&T, gaining all rights to the UNIX source code, for US$150 million. [14] (December 1993 [45.141]) (January 1993 [70.1]) (US$350 million [117.99])

December 31

- Workstation market share for the year: HP PA-RISC 31%, Sun SPARC 25%, MIPS 20%, IBM RS/6000 12%. [183]

End of 1991-1992. Next: 1993.

1968-1986 | 1987-1990 | **1991-1992** | 1993 | 1994 | 1995-end

This complete timeline is provided on the Web for free. If you wish to contribute money to help my keeping this timeline available and updated, I would very much appreciate it.

A list of **references** to all source material is available.

Also check my list of other timelines.

Last updated: 2005 April 29.

URL=http://www.islandnet.com/~kpolsson/workstat/

Link to Ken P's home page.

JDJ The World's Leading Java Resource

A Reusable Drag and Drop Handler

The ability to transfer information by dragging data from one component to another has been around since the development of the graphical user interface. Over the years drag-and-drop has gone from a cool feature to a required piece of most user interfaces. Most users expect to be able to drag objects between fields, windows, or folders and have some action occur. Drag-and-drop is even used to open applications by dragging a file to an application icon.

For a Java developer creating user interfaces it's no longer a question of *whether* drag-and-drop should be used but of *how much.* Java provides a set of classes for implementing the drag-and-drop interface. While it's not overly complex, implementing it in a complex GUI with a number of drag sources and drop targets can be tedious and error-prone. In this article, I develop an abstract DnDHandler class that takes care of most of the tedium and simplifies the implementation of drag-and-drop.

Drag-and-Drop Review

Although there are many parts to a drag-and-drop transaction, it can basically be broken down into three main components (see Figure 1):

1. Starting the drag where the drag action is recognized by the component
2. Converting the drag item into a transferable data type
3. Dropping the transferable data into the drop target

During the drag-and-drop transaction many other minor parts allow for user feedback, but these won't be discussed in any detail in this article. For more information on the details of drag-and-drop, see the reference at the end of the article.

Consider the simple drag-and-drop application shown in Figure 2 that contains two components: a DraggableTree on the left and a DroppableList on the right. The user can select items from the tree and drag them into the list.

The code for the DraggableTree is shown in Listing 1 and the code for the DroppableList is shown in Listing 2. Listings 1-5 can be downloaded from www.sys-con.com/java/sourcec.cfm.

To make the tree a draggable component a number of things must be done. First, the tree must implement the DragGestureListener interface, then create an instance of a DragSource and call the createDefaultDragGestureRecognizer method so that the tree will be notified when a drag action has been initiated.

When a drag-and-drop action occurs, the dragGestureRecognized method is called. This method first checks to see if something has been selected. If it has, the method then gets the selected object, creates a transferable version of the data, and calls the start drag function.

The list that acts as the drop target must implement the DropTargetListener interface and create a

DropTarget instance. The DropTarget constructor is used to notify the drag-and-drop framework that the list will accept dragged objects.

Although a number of methods are defined in the DropTargetListener interface, the main one is the drop method, which is called when an object is dropped into the list.

The drop method gets the transferable data from the DropTargetDropEvent. If the transferable data is the right data type, a number of steps are taken. First, the drop is accepted. Next, the transferred data is retrieved and the item is added to the list. Last, the dropComplete function is called to notify the drag-and-drop framework that the drop was completed successfully.

There's a lot more to drag-and-drop than presented in this simple example, but the example shows the basic steps in any drag-and-drop transaction.

The DNDHandler Class

Unlike the previous example, implementing drag-and-drop in a more complex GUI or a multiple document interface involves a lot of code duplication, if there are more than a couple of drag-and-drop targets. It's better if all the common code for a drag-and-drop transaction is contained in a single class where it's reused by any GUI component that needs to implement drag-and-drop.

This common drag-and-drop class is called *DnDHandler*. It should contain as much of the drag-and-drop functionality as possible and implement both the drag and the drop interfaces.

The requirements needed to create a draggable GUI component are:

- Implement the DragGestureListener interface
- Create an instance of a DragSource
- Call the createDefaultDragGestureRecognizer method so the component will be notified when a drag action has been initiated
- Create a dragGestureRecognized method that will get the draggable data and start the drag

The only problem with making these requirements into a generic class is that the dragGestureRecognized method must know how to get data from the actual component. To get around this problem, the dragGestureRecognized in the DnDHandler class is changed so it calls a getTransferable method to get the data from the actual component. The DnDHandler class makes this an abstract method so it will need to be implemented by the class extending it.

Now let's look at the requirements for a droppable GUI component:

- Implement the DropTargetListener interface
- Create an instance of a DropTarget instance
- Create a drop method to add the dragged data to the component

Again, the only problem with making these requirements generic is the drop method, which needs to know how to add the data to the dropped component. For the DnDHandler class we'll modify the drop method to call an abstract handleDrop method that needs to be implemented by the class extending the DnDHandler class. The resulting DnDHandler class is shown in Listing 3.

The DnDHandler class implements the DropTargetListener, DragSourceListener, and DragGestureListener interfaces. Its constructor takes the components as an argument and creates an instance of a DragSource and DropTarget and contains three abstract methods:

1. ***getTransferable:*** Gets transferable data from the component
2. ***handleDrop:*** Handles the drop action
3. ***getSupportedDataFlavors:*** Gets the transferable data that's supported

This DnDHandler class encapsulates all the requirements needed to create a drag-and-drop interface.

Using the DnDHandler Class

Let's now rework our original drag-and-drop example using the DnD- Handler class and see how this simplifies the implementation. To use the DnDHandler class, the new class that's used to replace DraggableTree needs to extend both JTree and the DnDHandler classes, but Java's single inheritance limitation makes this impossible. To get around this limitation we use an inner class to extend the DnDHandler class, while the main class extends JTree.

The reworked DraggableTree class, DNDTree, is shown in Listing 4. The inner class DNDTreeHandler does most of the work. The only thing the outer class does is create an instance of the DNDTreeHandler. The DNDTreeHandler class has the one argument constructor needed by its parent and the implementation of the three abstract methods.

By implementing the handleDrop, the DNDTree class has some improved functionality over the original version. The tree is now a drop target and items from the list can be dragged onto the tree. This is one of the advantages of using the DNDHandler class: instead of implementing all the methods of a DropTargetListener, we simply write the handleDrop function. If we didn't want the DNDTree to be a drop target, we would still have to implement the handleDrop function, but we could just ignore the drop event or call the rejectDrop method.

The reworked Droppable list class is called DNDList (see Listing 5). It's coded in a similar manner to the DNDTree class.

The reworked example is still a simple application of drag-and-drop but it does show how the DNDHandler calls simplify drag-and-drop implementation. Instead of implementing three interfaces and numerous methods, drag-and-drop can now be executed by writing three simple methods.

Conclusion

The DnDHandler class has proven very useful in our development of user interfaces. It allows us to add drag-and-drop features to the user interface more quickly and also provides a central place to control the low-level workings of drag-and-drop. We're using it under Java 1.3 on both Windows and UNIX and haven't experienced any problems.

Most real-world projects will have many more transferable data types than the simple example presented. The management of these transferable data types is an important implementation issue that must be considered when using drag-and-drop in a complex user interface.

Hopefully this article has taken some of the mystery out of implementing drag-and-drop. Although

it looks difficult, it's fairly straightforward using a class such as DnDHandler.

Reference

1. Zukowski, J. (1999). *John Zukowski's Definitive Guide to Swing for Java 2.* Apress.

The Early History of Smalltalk

Alan C. Kay
Apple Computer
kay2@apple.com.Internet#

HOPL-II/4/93/MA, USA
© 1993 ACM 0-89791-571-2/93/0004/0069...$1.50

Abstract

Most ideas come from previous ideas. The sixties, particularly in the ARPA community, gave rise to a host of notions about "human-computer symbiosis" through interactive time-shared computers, graphics screens and pointing devices. Advanced computer languages were invented to simulate complex systems such as oil refineries and semi-intelligent behavior. The soon-to-follow paradigm shift of modern personal computing, overlapping window interfaces, and object-oriented design came from seeing the work of the sixties as something more than a "better old thing." This is, more than a better way: to do mainframe computing; for end-users to invoke functionality; to make data structures more abstract. Instead the promise of exponential growth in computing /$/ volume demanded that the sixties be regarded as "almost a new thing" and to find out what the actual "new things" might be. For example, one would computer with a handheld "Dynabook" in a way that would not be possible on a shared mainframe; millions of potential users meant that the user interface would have to become a learning environment along the lines of Montessori and Bruner; and needs for large scope, reduction in complexity, and end-user literacy would require that data and control structures be done away with in favor of a more biological scheme of protected universal cells interacting only through messages that could mimic any desired behavior.

Early Smalltalk was the first complete realization of these new points of view as parented by its many predecessors in hardware, language and user interface design. It became the exemplar of the new computing, in part, because we were actually trying for a qualitative shift in belief structures--a new Kuhnian paradigm in the same spirit as the invention of the printing press-and thus took highly extreme positions which almost forced these new styles to be invented.

Table of Contents

Introduction

I'm writing this introduction in an airplane at 35,000 feet. On my lap is a five pound notebook computer--1992's "Interim Dynabook"--by the end of the year it sold for under $700. It has a flat, crisp, high-resolution bitmap screen, overlapping windows, icons, a pointing device, considerable storage and computing capacity, and its best software is object-oriented. It has advanced networking built-in and there are already options for wireless networking. Smalltalk runs on this system, and is one of the main systems I use for my current work with children. In some ways this is more than a Dynaboo (quantitatively), and some ways not quite there yet (qualitatively). All in all, pretty much what was in mind

during the late sixties.

Smalltalk was part of this larger pursuit of ARPA, and later of Xerox PARC, that I called personal computing. There were so many people involved in each stage from the research communities that the accurate allocation of credit for ideas in intractably difficult. Instead, as Bob Barton liked to quote Goethe, we should "share in the excitement of discover without vain attempts to claim priority."

I will try to show where most of the influences came from and how they were transformed in the magnetic field formed by the new personal computing metaphor. It was the attitudes as well as the great ideas of the pioneers that helped Smalltalk get invented. Many of the people I admired most at this time--such as Ivan Sutherland, Marvin Minsky, Seymour Papert, Gordon Moore, Bob Barton, Dave Evans, Butler Lampson, Jerome Bruner, and others--seemed to have a splendid sense that their creations, though wonderful by relative standards, were not near to the absolute thresholds that had to be crossed. Small minds try to form religions, the great ones just want better routes up the mountain. Where Newton said he saw further by standing on the shoulders of giants, computer scientists all too often stand on each other's toes. Myopia is still a problem where there are giants' shoulders to stand on--"outsight" is better than insight--but it can be minimized by using glasses whose lenses are highly sensitive to esthetics and criticism.

Programming languages can be categorized in a number of ways: imperative, applicative, logic-based, problem-oriented, etc. But they all seem to be either an "agglutination of features" or a "crystallization of style." COBOL, PL/1, Ada, etc., belong to the first kind; LISP, APL-- and Smalltalk--are the second kind. It is probably not an accident that the agglutinative languages all seem to have been instigated by committees, and the crystallization languages by a single person.

Smalltalk's design--and existence--is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages. Philosophically, Smalltalk's objects have much in common with the monads of Leibniz and the notions of 20th century physics and biology. Its way of making objects is quite Platonic in that some of them act as idealisations of concepts--Ideas--from which manifestations can be created. That the Ideas are themselves manifestations (of the Idea-Idea) and that the Idea-Idea is a-kind-of Manifestation-Idea--which is a-kind-of itself, so that the system is completely self-describing-- would have been appreciated by Plato as an extremely practical joke [Plato].

In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than the whole--like data structures, procedures, and functions which are the usual paraphernalia of programming languages--each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computer all hooked together by a very fast network. Questions of concrete representation can thus be postponed almost indefinitely because we are mainly concerned that the computers behave appropriately, and are interested in particular strategies only if the results are off or come back too slowly.

Though it has noble ancestors indeed, Smalltalk's contribution is anew design paradigm--which I called object-oriented--for attacking large problems of the professional programmer, and making small ones possible for the novice user. Object-oriented design is

a successful attempt to qualitatively improve the efficiency of modeling the ever more complex dynamic systems and user relationships made possible by the silicon explosion.

"We would know what they thought
when the did it."
--Richard Hamming

"Memory and imagination are but two
words for the same thing."
--Thomas Hobbes

In this history I will try to be true to Hamming's request as moderated by Hobbes' observation. I have had difficulty in previous attempts to write about Smalltalk because my emotional involvement has always been centered on personal computing as an amplifier for human reach--rather than programming system design--and we haven't got there yet. Though I was the instigator and original designer of Smalltalk, it has always belonged more to the people who make it work and got it out the door, especially Dan Ingalls and Adele Goldberg. Each of the LRGers contributed in deep and remarkable ways to the project, and I wish there was enough space to do them all justice. But I think all of us would agree that for most of the development of Smalltalk, Dan was the central figure. Programming is at heart a practical art in which real things are built, and a real implementation thus has to exist. In fact many if not most languages are in use today not because they have any real merits but because of their existence on one or more machines, their ability to be bootstrapped, etc. But Dan was far more than a great implementer, he also became more and more of the designer, not just of the language but also of the user interface as Smalltalk moved into the practical world.

Here, I will try to center focus on the events leading up to Smalltalk-72 and its transition to its modern form as Smalltalk-76. Most of the ideas occurred here, and many of the earliest stages of OOP are poorly documented in references almost impossible to find.

This history is too long, but I was amazed at how many people and systems that had an influence appear only as shadows or not at all. I am sorry not to be able to say more about Bob Balzer, Bob Barton, Danny Bobrow, Steve Carr, Wes Clark, Barbara Deutsch, Peter Deutsch, Bill Duvall, Bob Flegal, Laura Gould, Bruce Horn, Butler Lampson, Dave Liddle, William Newman, Bill Paxton, Trygve Reenskaug, Dave Robson, Doug Ross, Paul Rovner, Bob Sproull, Dan Swinehart, Bert Sutherland, Bob Taylor, Warren Teitelman, Bonnie Tennenbaum, Chuck Thacker, and John Warnock. Worse, I have omitted to mention many systems whose design I detested, but that generated considerable useful ideas and attitudes in reaction. In other words, histories" should not be believed very seriously but considered as "FEEBLE GESTURES PFF" done long after the actors have departed the stage.

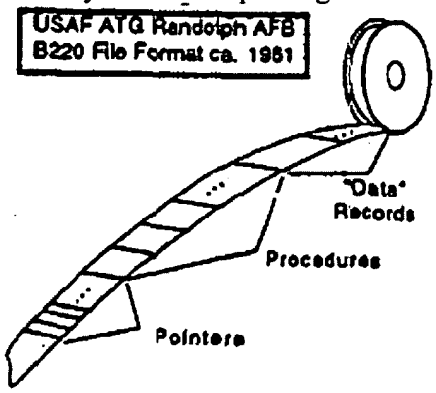
Thanks to the numerous reviewers for enduring the many drafts they had to comment on. Special thanks to Mike Mahoney for helping so gently that I heeded his suggestions and so well that they greatly improved this essay--and to Jean Sammet, an old old friend, who quite literally frightened me into finishing it--I did not want to find out what would happen if I were late. Sherri McLoughlin and Kim Rose were of great help in getting all the materials together.

I. 1960-66--Early OOP and other formative ideas of the sixties

Though OOP came from many motivations, two were central. The large scale one was to find a better module scheme for complex systems involving hiding of details, and the small scale one was to find a more flexible version of assignment, and then to try to eliminate it altogether. As with most new ideas, it originally happened in isolated fits and starts.

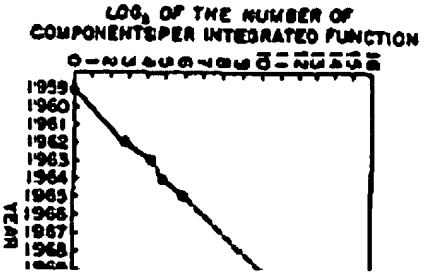
New ideas go through stages of acceptance, both from within and without. From within, the sequence moves from "barely seeing" a pattern several times, then noting it but not perceiving its "cosmic" significance, then using it operationally in several areas, then comes a "grand rotation" in which the pattern becomes the center of a new way of thinking, and finally, it turns into the same kind of inflexible religion that it originally broke away from. From without, as Schopenhauer noted, the new idea is first denounced as the work of the insane, in a few years it is considered obvious and mundane, and finally the original denouncers will claim to have invented it.

True to the stages, I "barely saw" the idea several times ca. 1961 while a programmer in the Air Force. The first was on the Burroughs 220 in the form of a style for transporting files from one Air Training Command installation to another. There were no standard operating systems or file formats back then, so some (t this day unknown) designer decided to finesse the problem by taking each file and dividing it into three parts. The third part was all of the actual data records of arbitrary size and format. The second part contained the B220 procedures that knew how to get at records and fields to copy and update the third part. And the first part was an array or relative pointers into entry points of the procedures in the second part (the initial pointers were in a standard order representing standard meanings). Needless to say, this was a great idea, and was used in many subsequent systems until the enforced use of COBOL drove it out of existence.



The second barely-seeing of the idea came just a little later when ATC decided to replace the 220 with a B5000. I didn't have the perspective to really appreciate it at the time, but I did take note of its segmented storage system, its efficiency of HLL compilation and byte-coded execution, its automatic mechanisms for subroutine calling and multiprocess switching, its pure code for sharing, its protected mechanisms, etc. And, I saw that the access to its Program Reference Table corresponded to the 220 file system scheme of providing a procedural interface to a module. However, my big hit from this machine at this time was not the OOP idea, but some insights into HLL translation and evaluation. [Barton, 1961] [Burroughs, 1961]

After the Air Force, I worked my way through the rest of college by programming mostly retrieval systems for large collections of weather data for the National Center for Atmospheric Research. I got interested in simulation in general--particularly of one machine by another--but aside from doing a one-dimensional version of a bit-field block transfer (bitblt) on a CDC 6600 to simulate word sizes of various machines, most of my attention was distracted by school, or I should say the theatre at school. While in Chippewa Falls helping to debug the 6600, I read an article by Gordon Moore which



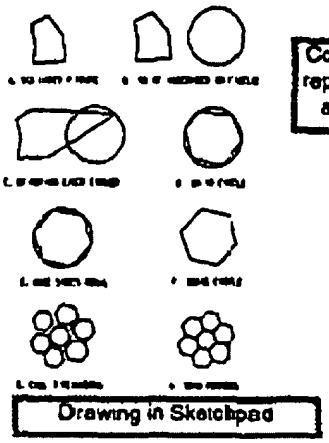
predicted that integrated silicon on chips was going to exponentially improve in density and cost over many years [Moore 65]. At the time in 1965, standing next to the room-sized freon-cooled 10 MIP 6600, his astounding predictions had little projection into my horizons.

Sketchpad and Simula

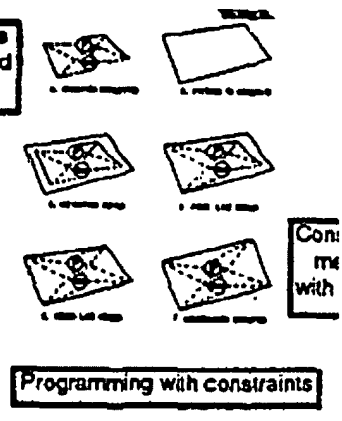
Through a series of flukes, I wound up in graduate school at the University of Utah in the Fall of 1966, "knowing nothing." That is to say, I had never heard of ARPA or its projects, or that Utah's main goal in this community was to solve the "hidden line" problem in 3D graphics, until I actually walked into Dave Evans' office looking for a job and a desk. On Dave's desk was a foot-high stack of brown covered documents, one of which he handed to me: "Take this and read it."

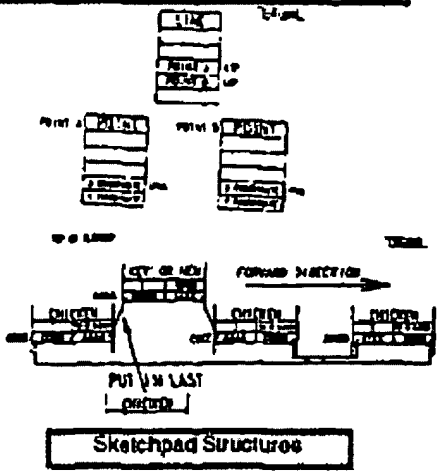
Every newcomer got one. The title was "Sketchpad: A man-machine graphical communication system" [Sutherland, 1963]. What it could do was quite remarkable, and completely foreign to any use of a computer I had ever encountered. The three big ideas that were easiest to grapple with were: it was the invention of modern interactive computer graphics; things were described by making a "master drawing" that could produce "instance drawings"; control and dynamics were supplied by "constraints," also in graphical form, that could be applied to the masters to shape an inter-related parts. Its data structures were hard to understand--the only vaguely familiar construct was the embedding of pointers to procedures and using a process called reverse indexing to jump through them to routines, like the 22- file system [Ross, 1961]. It was the first to have clipping and zooming windows--one "sketched" on a vitual sheet about 1/3 mile square!

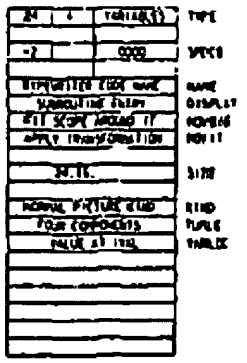

When there was only one personal computer
Ivan at the TX-2 ca. 1962

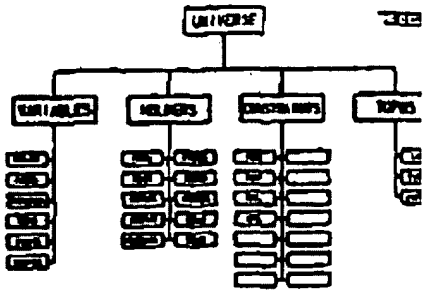


Drawing in Sketchpad


Programming with constraints


Sketchpad Structures


"generic block" showing procedural attachment


Sketchpad's "inheritance" hierarchy

Head whirling, I found my desk. ON it was a pile of tapes and listings, and a note: "This is the Algol for the 1108. It doesn't work. Please make it work." The latest graduate student gets the latest dirty task.

The documentation was incomprehensible. Supposedly, this was the Case-Western Reserve 1107 Algol--but it had been doctored to make a language called Simula; the documentation read like Norwegian transliterated into English, which in fact it was. There were uses of words like *activity* and *process* that didn't seem to coincide with normal English usage.

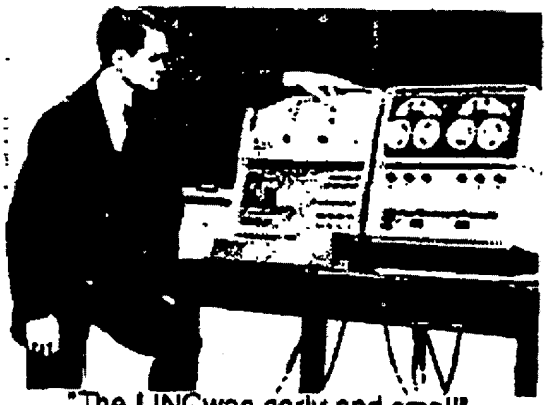
Finally, another graduate student and I unrolled the program listing 80 feet down the hall and crawled over it yelling discoveries to each other. The weirdest part was the storage allocator, which did not obey a stack discipline as was usual for Algol. A few days later, that provided the clue. What Simula was allocating were structures very much like the instances of Sketchpad. There wee descriptions that acted like masters and they could create instances, each of which was an independent entity. What Sketchpad called masters and instances, Simula called activities and processes. Moreover, Simula was a procedural language for controlling Sketchpad-like objects, thus having considerably more flexibility than constraints (though at some cost in elegance) [Nygaard, 1966, Nygaard, 1983].

This was the big hit, and I've not been the same since. I think the reason the hit had such impact was that I had seen the idea enough times in enough different forms that the final recognition was in such general terms to have the quality of an epiphany. My math major had centered on abstract algebras with their few operations generally applying to many

structures. My biology manor had focused on both cell metabolism and larger scale morphogenesis with its notions of simple mechanisms controlling complex processes and one kind of building block able to differentiate into all needed building blocks. The 220 file system, the B5000, Sketchpad, and finally Simula, all used the same idea for different purposes. Bob Barton, the main designer of the B5000 and a professor at Utah had said in one of his talks a few days earlier: "The basic principal of recursive design is to make the parts have the same power as the whole." For the first time I thought of the whole as the entire computer and wondered why anyone would want to divide it up into weaker things called data structures and procedures. Why not divide it up into little computers, as time sharing was starting to? But not in dozens. Why not thousands of them, each simulating a useful structure?

I recalled the monads of Leibniz, the "dividing nature at its joints" discourse of Plato, and other attempts to parse complexity. Of course, philosophy is about opinion and engineering is about deeds, with science the happy medium somewhere in between. It is not too much of an exaggeration to say that most of my ideas from then on took their roots from Simula--but not as an attempt to improve it. It was the promise of an entirely new way to structure computations that took my fancy. As it turned out, it would take quite a few years to understand how to use the insights and to devise efficient mechanisms to execute them.

II. 1967-69--The FLEX Machine, a first attempt at an OOP-based personal computer



"The LINCwas early and small"
Wes Clark and the LINC, ca 1962

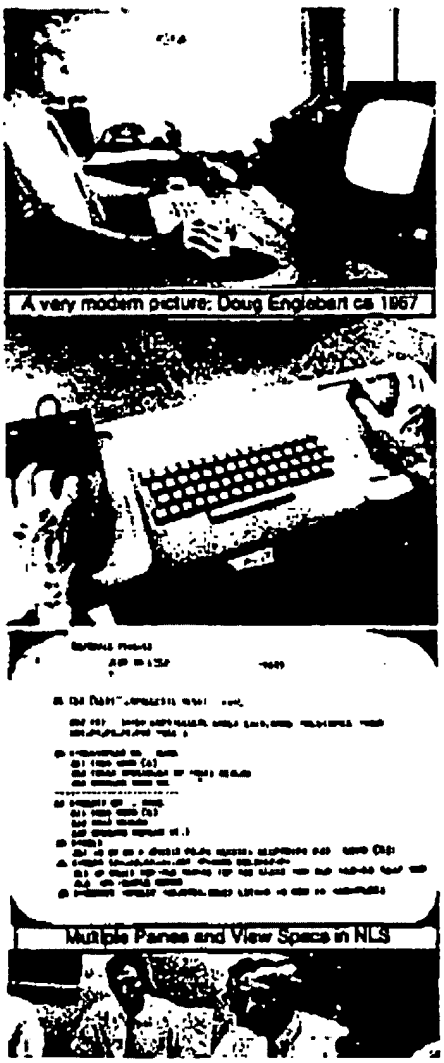
Dave Evans was not a great believer in graduate school as an institution. As with many of the ARPA "contracts" he wanted his students to be doing "real things"; they should move through graduate school as quickly as possible; and their theses should advance the state of the art. Dave would often get consulting jobs for his students, and in early 1967, he introduced me to Ed Cheadle, a friendly hardware genius at a local aerospace company who was working on a "little machine." It was not the first personal computer--that was the LINC of Wes Clark--but Ed wanted it for noncomputer professionals, in particular, he wanted to program it in a higher level language, like BASIC. I said; "What about JOSS? It's nicer." He said: "Sure, whatever you think," and that was the start of a very pleasant collaboration we called the FLEX machine. As we jot deeper into the design, we realized that we wanted to dynamically simulate and extend, neither of which JOSS (or any existing language that I knew of) was particularly good at. The machine was too small for Simula, so that was out. The beauty of JOSS was the extreme attention of its design to the end-user--in this respect, it has not been surpassed [Joss 1964, Joss 1978]. JOSS was too slow for serious computing (but cf. Lampson 65), did not have real procedures, variable scope, and so forth. A language that looked a little like JOSS but had considerably more potential power was Wirth's EULER [Wirth 1966]. This was a generalization of Algol along lines first set forth by van Wijngaarden [van Wijngaarden 1963] in which types were discarded, different features consolidated, procedures were made into first class objects, and so forth. Actually kind of LISPlike, but without the deeper insights of LISP.

But EULER was enough of "an almost new thing" to suggest that the same techniques be applied to simply Simula. The EULER compiler was a part of its formal definition and made a simple conversion into 85000-like byte-codes. This was appealing because it s suggested the Ed's little machine could run byte-codes emulated in the longish slow microcode that was then possible. The EULER compiler however, was tortuously rendered in an "extended precedence" grammar that actually required concessions in the language syntax (e.g. "," could only be used in one role because the precedence scheme had no state space). I initially adopted a bottom-up Floyd-Evans parser (adapted from Jerry Feldman's original compiler-compiler [Feldman 1977]) and later went to various top-down schemes, several of them related to Shorre's META II [Shorre 1963] that eventually put the translater in the name space of the language.

The semantics of what was now called the FLEX language needed to be influenced more by Simula than by Algol or EULER. But it was not completely clear how. Nor was it clear how the users should interact with the system. Ed had a display (for graphing, etc.) even on his first machine, and the LINC had a "glass teletype," but a Sketchpad-like system seemed far beyond the scope that we could accomplish with the maximum of 16k 16-bit words that our cost budget allowed.

Doug Engelbart and NLS

This was in early 1967, and while we were pondering the FLEX machine, Utah was visited by Doug Engelbart. A prophet of Biblical dimensions, he was very much one of the fathers of what on the FLEX machine I had started to call "personal computing." He actually traveled with his own 16mm projector with a remote control for starting a and stopping it to show what was going on (people were not used to seeing and following cursors back then). His notion on the ARPA dream was that the destiny of Online Systems (MLS) was the "augmentation of human intellect" via an interactive vehicle navigating through "thought vectors in concept space." What his system could do then--even by today's standards--was incredible. Not just hypertext, but graphics, multiple panes, efficient navigation and command input, interactive collaborative work, etc. An entire conceptual world and world view [Engelbart 68]. The impact of this vision was to produce in the minds of those who were "eager to be augmented" a compelling metaphor of what interactive computing should be like, and I immediately adopted many of the ideas for the FLEX machine.



In the midst of the ARPA context of human-computer symbiosis and in the presence of Ed's "little machine", Gordon Moore's "Law" again came to mind, this time with great impact. For the first time I made the leap of putting the room-sized interactive TX-2 or even a 10 MIP 6600 on a desk. I was almost frightened by the implications; computing as we knew it couldn't survive--

the actual meaning of the word changed--it must have
been the same kind of disorientation people had after reading Copernicus and first looked up from a different Earth to a different Heaven.

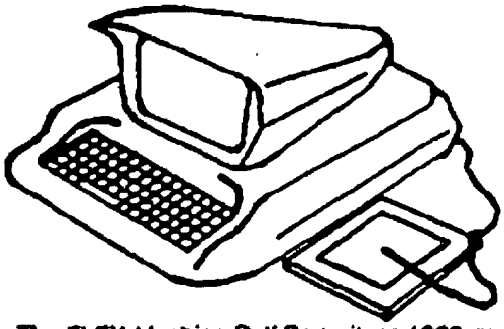
Instead of at most a few thousand institutional mainframes in the world--even today in 1992 it is estimated that there are only 4000 IBM mainframes in the entire world--and at most a few thousand users trained for each application, there would be millions of personal machines and users, mostly outside of direct institutional control. Where would the applications and training come from? Why should we expect an applications programmer to anticipate the specific needs of a particular one of the millions of potential users? An extensional system seemed to be called for in which the end-users would do most of the tailoring (and even some of the direct constructions) of their tools. ARPA had already figured this out in the context of their early successes in time-sharing. Their larger metaphor of human-computer symbiosis helped the community avoid making a religion of their subgoals and kept them focused on the abstract holy grail of "augmentation."

One of the interested features of NLS was that its user interface was a parametric and could be supplied by the end user in the form of a "grammar of interaction given in their compiler-compiler TreeMeta. This was similar to William Newman's early "Reaction Handler" [Newman 66] work in specifying interfaces by having the end-user or developer construct through tablet and stylus an iconic regular expression grammar with action procedures at the states (NLS allowed embeddings via its context free rules). This was attractive in many ways, particularly William's scheme, but to me there was a monstrous bug in this approach. Namely, these grammars forced the user to be in a system state which required getting out of before any new kind of interaction could be done. In hierarchical menus or "screens" one would have to backtrack to a master state in order to go somewhere else. What seemed to be required were states in which there was a transition arrow to every other state--not a fruitful concept in formal grammar theory. In other words, a much "flatter" interface seemed called for--but could such a thing be made interesting and rich enough to be useful?

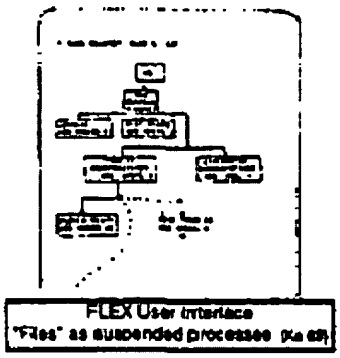
Again, the scope of the FLEX machine was too small for a miniNLS, and we were forced to find alternate designs that would incorporate some of the power of the new ideas, and in some cases to improve them. I decided that Sketchpad's notion of a general window that viewed a larger virtual world was a better idea than restricted horizontal panes and with Ed came up with a clipping algorithm very similar to that under development at the same time by Sutherland and his students at Harvard for the 3D "virtual reality" helment project [Sutherland 1968].

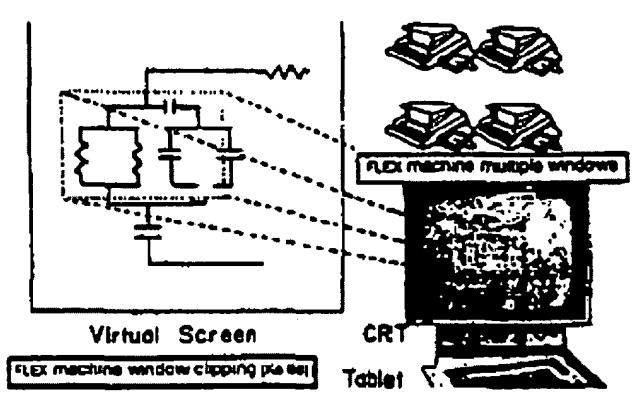
Object references were handled on the FLEX machine as a generalization of B5000 descriptors. Instead of a few formats for referencing numbers, arrays, and procedures, a FLEX descriptor contained two pointers: the first to the "master" of the object, and the second to the object instances (later we realized that we should put the master pointer in the instance to save space). A different method was taken for handling generalized assignment. The B5000 used l-values and r-values [Strachey*] which worked for some cases but couldn't handle more complex objects. For example: a[55] := 0 if a was a sparse array whose default element was - would still generate an element in the array because := is an "operator" and a[55] is dereferenced into an l-value before anyone gets to see that the r-value is the default element, regardless of whether a is an array or a procedure fronting for an array. What is needed is something like: a(55 := 0), which can look at all relevant operands before any store is made. In other words, := is not an operator, but a kind of a

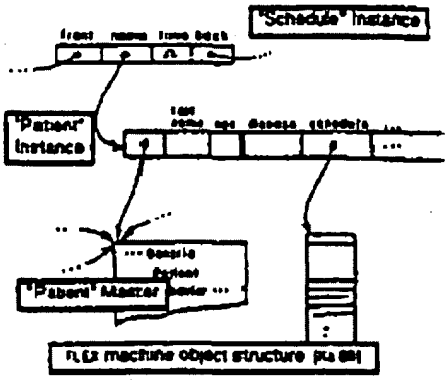
index that can select a behavior from a complex object. It took me a remarkably long time to see this, partly I think because one has to invert the traditional notion of operators and functions, etc., to see that objects need to privately own all of their behaviors: *that objects are a kind of mapping whose values are its behaviors*. A book on logic by



The FLEX Machine Self Portrait, ca 1968 (Ka 69)







Carnap [Ca *] helped by showing that "intentional" definitions covered the same territory as the more traditional extensional technique and were often more intuitive and convenient.

As in Simula, a coroutine control structure [Conway, 1963] was used as a way to suspend and resume objects. Persistent objects like files and documents were treated as suspended processes and were organized according to their Algol-like static variable scopes. These were shown on the screen and could be opened by pointing at them. Coroutining was also used as a control structure for looping. A single operator while was used to test the generators which returned false when unable to furnish a new value. Booleans were used to link multiple generators. So a "for-type" loop would be written as:

```
while i <= 1 to 30 by 2 ^ j <= 2 to k by 3 do j<-j * i;
```

where the ... to ... by ... was a kind of coroutine object. Many of these ideas were reimplemented in a stronger style in Smalltalk later on.

Another control structure of interest in FLEX was a kind of event-driven "soft interrupt" called **when**. Its boolean expression was compiled into a "tournement soft" tree that cached all possible intermediate results. The relevant variables were threaded through all of the sorting trees in all of the whens so that any change only had to compute through the necessary parts of the booleans. The efficiency was very high and was similar to the techniques now used for spreadsheets. This was an embarrassment of riches with difficulties often encountered in event-driven systems. Namely, it was a complex task to control the

context of just when the whens should be sensitive. Part of the boolean expression had to be used to check the contexts, where I felt that somehow the structure of the program should be able to set and unset the event drivers. This turned out to beyond the scope of the FLEX system and needed to wait for a better architecture.

Still, quite a few of the original FLEX ideas in their proto-object form did turn out to be small enough to be feasible on the machine. I was writing the first compiler when something unusual happened: the Utah graduate students got invited to the ARPA contractors meeting held that year at Alta, Utah. Towards the end of the three days, Bob Taylor, who had succeeded Ivan Sutherland as head of ARPA-IPTO asked the graduate students (sitting in a ring around the outside of the 20 or so contractors) if they had any comments. John Warnock raised his hand and pointed out that since the ARPA grad students would all soon be colleagues (and since we did all the real work anyway), ARPA should have a contractors-type meeting each user for the grad students. Taylor thought this was a great idea and set it up for the next summer.

Another ski-lodge meeting happened in Park City later that spring. The general topic was education and it was the first time I heard Marvin Minsky speak. He put forth a terrific diatribe against traditional education methods, and from him I heard the ideas of Piaget and Papert for the first time. Marvin's talk was about how we think about complex situations and why schools are really bad places to learn these skills. He didn't have to make any claims about computer+kids to make his point. It was clear that education and learning had to be rethought in the light of 20th century cognitive psychology and how good thinkers really think. Computing enters as a new representation system with new and useful metaphors for dealing with complexity, especially of systems [Minsky 70].

For the summer 1968 ARPA grad students meeting at Allerton House in Illinois, I boiled all the mechanisms in the FLEX machine down into one 2'x3' chart. This included all the "object structures." the compiler, the byte-code interpreter, i/o handlers, and a simple display editor for text and graphics. The grad students were a distinguished group that did indeed become colleagues in subsequent years. My FLEX machine talk was a success, but the big whammy for me came during a tour of U of Illinois whee I saw a 1" square lump of class and neon gas in which individual spots would light up on command--it was the first flat-panel display. I spent the rest of the conference calculating just when the silicon of the FLEX machine could be put on the back of the display. According to Gordon Moore's "Law", the answer seemed to be sometime in the late seventies or early eighties. A long time off--it seemed to long to worry much about it then.

But later that year at RAND I saw a truly beautiful system. This was GRAIL, the graphical followin to JOSS. The first tablet (the famous RAND tablet) was invented by Tom Ellis [Davvis 1964] in order to capture human gestures, and Gave Groner wrote a program to eficiently recognize and respnd to them [Groner 1966]. Through everything was fastned with bubble gum and the stem crashed often, I have never forgotton my fist interactions with this system. It was direct manipulation, it was analogical, it was modeless, it was beautiful. I reallized that the FLEX interface was all wrong, but how could something like GRAIOL be stuffed intosuch a tiny machine since it required all of a stand-alone 360/44 to run in?

A month later, I finally visited Semour Papert, Wally Feurzig, Cynthia Solomon and some of the other original reserachers who had built LOGO and were using it with children in the Lexington schools. Here were children doing real pogramming with a specially designed

language and environment. As with Simulas leading to OOP, this enoucnter final hit me with what the destiny of personal computing *really* was going to be. Not a personal dynamic vehicle, as in Engelbart's metaphor opposed to the IBM "railroads", but something much more profound: a personal dynamic medium. With a vehicle on could wait until high school and give "drivers ed", but if it was a medium, it had to extend into the world of childhood.

Now the collision of the FLEX machine, the flat-screen display, GRAIL, Barton's "communications" talk, McLuhan, and Papert's work with children all came together to form an image of what a personal computer really should be. I remembered Aldus Manutius who 40 years after the printing press put the book into its modern dimensions by making it fit into saddlebags. It had to be no larger than a notebook, and needed an interface as friendly as JOSS', GRAIL's, and LOGO's, but with the reach of Simula and FLEX. A clear romantic vision has a marvelous ability to focus thought and will. Now it was easy to know what to do next. I built a cardboard model of it to see what if would look and feel like, and poured in lead pellets to see how light it would have to be (less than two pounds). I put a keyboard on it as well as a stylus because, even if handprinting and writing were recognized perfectly (and there was no reason to expect that it would be), there still needed to be a blance between the lowspeed tactile degrees of freedom offered by the stylus and the more limited but faster keyboard. Since ARPA was starting to experiment with packet radio, I expected that the Dynabook when it arrived a decade or so hence, wouldhave a wireless networking system.

Early next year (1969) there was a conference on Extensible Languages in which alnost every famous name in the field attended. The debate was great and wighty--it was a religious war of unimplemented poorly though out ideas. As Alan Perlis, one of the great men in Computer Science, put it with characteristic wit:

> It has been such a long time since aI have seen so many familiar faces shouting among so many familiar ideas. Discover of something new in programming languages, like any discovery, has somewhat the same sequence of emotions as falling in love. A sharp eleation followed by euphoria, a feeling of uniuqeness, and ultimately the wandering eye (the urge to generalize) [ACM 69].

But it was all talk--no one had done anything yet. In the midst of all this, Ned Irons got up and presented IMP, a system that had already been working for several years that was more elegant than most of the nonworking proposals. The basic idea of IMP was that you coulduse any phrase in the grammar as a procedur heading and write a semantic definition in terms of the language as extended so far [Irons 1970].

I had already made the first version of the FLEX machine syntax driven, but where the meaning of a phrase was defned in the more usual way as the kind of code that was emitted. This separated the compiiler-extensor part of the system from the end-user. In Irons' approach, every procedure in the system define dits own syntax in a natural and useful manner. I cinorporated these ideas into the second verions of the FLEX machine and started to experiment with the idea of a direct interpreter rather than a syntax directed compiler. Somewhere in all of this, I realized that the bridge to an object-based system could be in terms of each object as a syntax directed interpreter of messages sent to it. In one fell swoop this would unify object-oriented semantics with the ideal of a completely extensible language. The mental image was one of separate computers sending requests to other computers that had to be accepted and understood by the receivers beofre anything could

happen. In todya's terms every object would be a *server* offering *services* whose deployment and discretion depended entirely on the server's notion of relationsip with the servee. As Liebniz said: "To get everything out of nothing, you only need to find one principle." This was not well thought out enough to do the FLEX machine any good, but formed a good point of departure for my thesis [Kay 69], which as Ivan Sutherland liked to say was "anything you can get three people to sign."

After three people signed it (Ivan was one off them), I went to the Stanford AI project and spent much more time thinking about notebook KiddyKomputers than AI. But there were two AI designs that were very intriguing. The first was Carl Hewitt's PLANNER, a programmable logic system that formed the deductive basis of Winograd's SHRDLU [Sussman 69, Hewitt 69] I designed several languages based on a combination of the pattern matching schemes of FLEX and PLANNER [Kay 70]. The second design was Pat Winston's concept formation system, a scheme for building semantic networks and comparing them to form analogies and learning processes [Winston 70]. It was kind of "object-oriented". One of its many good ieas was that the arcs of each net which served as attributes in AOV triples should themsleves be modeled as nets. Thus, for example a first order arc called LEFT-OF could be asked a higher order questions such as "What isyour converse?" and its net could answer: RIGHT-OF. This point of view later formed the basis for Minsky's frame systems [Minsky 75]. A few years later I wished I had paid more attention to this idea.

That fall, I heard a wonderful talk by Butler Lampson about CAL-TSS, a capability-based operating system that seemed very "object-oriented" [Lampson 69]. Unfogable pointers (ala 85000) were extended by bit-masks that restriected access to the object's internal operations. This confirmed my "objects as server" metaphor. There was also a very nice approach to exception handling which reminded me of the way failure was often handled in pattern matching systems. The only problem-- which the CAL designers did not wsee as a problam at all--was that only certain (usually large and slow) things were "objects". Fast things and small things, etc., weren't. This needed to be fixed.

The biggest hit for me while at SAIL in late '69 was to *really understand LISP*. Of course, every student knew about *car, cdr*, and *cons*, but Utah was impoverished in that no one there used LISP and hence, no one had penetrated thye mysteries of *eval* and *apply*. I could hardly believe how beautiful and wonderful the *idea* of LISP was [McCarthy 1960]. I say it this way because LISP had not only been around enough to get some honest barnacles, but worse, there wee deep falws in its logical foundations. By this, I mean that the pure language was supposed to be based on functions, but its most important components---such as lambda expressions quotes, and conds--where not functions at all, and insted ere called special forms. Landin and others had been able to get quotes and cons in terms of lambda by tricks that were variously clever and useful, but the flaw remained in the jewel. In the practical language things were better. There were not just EXPRs (which evaluated their arguments0, but FEXPRs (which did not). My next questions was, why on earth call it a functional language? Why not just base everuything on FEXPRs and force evaluation on the receiving side when needed? I could never get a good answer, but the question was very helpful when it came time to invent Smalltalk, because this started a line of thought that said "take the hardest and most profound thing you need to do, make it great, an then build every easier thing out of it". That was the promise of LiSP and the lure of lambda--needed was a better "hardest and most profound" thing. Objects should be it.

III. 1970-72--Xerox PARC: The KiddiKomp, miniCOM, and Smalltalk-71

In July 1970, Xerox, at the urgin of its chief scientist Jack Goldman, decdided to set up a long range reserach center in Palo Alo, California. In September, George Pake, the former chancellor at Washington University where Wes Clark's ARPA project was sited, hired Bob Taylor (who had left the ARPA office and was taling a sabbatical year at Utah) to start a "Computer Science Laboratory." Bob visited Palo Alto and we stayed up all night talking about it. The mansfield Amendment was threatening to blinkdly muzzle the most enlightened ARPA funding in favor of directly military reserach, and this new opportunity looked like a promising alternative. But work for a company? He wanted me to consult and I asked for a direction. He said: follow your instincts. I immediately started working up a new versio of the KiddiKimp tha could be made in enough quantity to do experiments leading to the user interface design for the eventual notebook. Bob Barton liked to say that "good ideas don't often scale." He was certainly right when applied to the FLEX machine. The B5000 just didn't directly scale down into a tiny machine. Only the byte-codes did. and even these needed modification. I decided to take another look at Wes Clark's LINKX, and was ready to appreciate it much more this time [Clark 1965].

I still liked pattern-directed approaches and OOP so I came up with a language design called "Simulation LOGO" or SLOGO for short *(I had a feeling the first versions migh run nice and slow). This was to be built into a SONY "tummy trinitron" and ould use a coarse bit-map display and the FLEX machine rubber tablet as a pointing device.

Another beautiful system that I had come across was Petere Deutsch's PDP-1 LISP (implemented when he was only 15) [Deutsch 1966]. It used onl 2K (18-bit words) of code and could run quite well in a 4K mahcine (it was its own operating system and interface). It seemed that even more could be done if the system were byte-coded, run by an architectural that was hoospitable to dynamic systems, and stuck into the ever larger ROMs that were becoming available. One of the basic insights I had gotten from Seymour was that you didn't have to do a lot to make a computer an "object for thought" for children, but what you did had to be done well and be able to apply deeply.

Right after New Years 1971, Bob Taylor scored an enourmous coup by attracting most of the struggling Berkeley computer corp to PARC. This group included Butler Lampson, Check Thacker, Peter Deutsch, Jim Mitchell, Dick Shoup, Willie Sue Haugeland, and Ed Fiala. Him Mitchell urged the group to hire Ed McCreight from CM and he arrived soon after. Gar Starkweather was there already, having been thrown out of the Xerox Rochester Labs for wanting to build a laser printer (which was against the local religion). Not long after, many of Doug Englebart's people joined up--part of the reason was that they want to reimplement NLS as a distributed network system, and Doug wanted to stay with time-sharing. The group included Bill English (the co-inventor of the mouse), Jeff Rulifson, and Bill Paxton.

Almost immediately we got into trouble with Xerox when the group decided that the new lab needed a PDP-10 for continuity with the ARPA community. Xerox (which has bought SDS essentially sight unseend a few years before) was horrified at the idea of their main compeititor's computer being used in the lab. They balked. The newly formed PARC group had a metting in which it was decided that it would take about three years to do a good operating system for the XDS SIGMA-7 but that we could build "our own PDP-10" in a

year. My reactopn was "Holy cow!" In fact, they pullit it off with considerable pnache. MAXC was actually a microcoded emeulation of the PDP-10 that used for the first time the new integrated chip memeoris (1K bits!) instead of core memory. Having practicalin house experience with both of these new technologies was critical for the more radical systems to come.

One little incident of LISP eauty happened when Allen Newell visited PARC with his theory of hierarchical thinking and was challenged to prove it. He was given a programming problem to solve while the protocol was collected. The problem was: given a list of items, produce a list consisteing of all of the odd indexed items followed by all of the even indexed items. Newel's internal programming langage resembple IPL-V in which pointers are manipulated explicitly, and he got into quite a struggle to do the program. In 2 seconds I wrote down:

```
oddsEvens(x) = append(odds(x), evens(x))
```

the statement of the problem in Landin's LISP syntax--and also the first part of the solution. Then a few seconds later:

```
where odds(x) = if null(x) v null(tl(x)) then x
                else hd(x) & odds(ttl(x))
      evens(x) = if null(x) v null(tl(x)) then nil
                 else odds(tl(x))
```

This characteristic of writing down many solutions in declarative form and have them also be the programs is part of the appeal and beauty of this kind of language. Watching a famous guy much smarter than I struggle for more than 30 minutes to not quite solve the problem his way (there was a bug) made quite an impression. It brought home to me once again that "point of view is worth 80 IQ points." I wasn't smarter but I had a much better internal thinking tool to amplify my abilities. This incident and others like it made paramount that any tool for children should have great thinking patterns and deep beeauty "built-in."

Right around this time we were involved in another conflict with Xerox management, in particular with Don Pendery the head "planner". He really didn't understand what we were talking about and instead was interested in "trends" and "what was the future going to be like" and how could Xerox "defend against it." I got so upset I said to him, "Look. The best way to predict the future is to invent it. Don't worry about what all those other people might do, this is the century in which almost any clear vision can be made!" He remained unconvinced, and that led to the famous "Pendery Papers for PARC Planning Purposese," a collection of essays on various aspects of the future. Mine proposed a version of the notebook as a "Display Transducer." and Jim Mitchell's was entitled "NLS on a Minicomputer."

Bill English took me under his wing and helped me start my group as I had always been a lone wolf and had no idea how to do it. One of his suggestions was that I should make a budget. I'm afraid that I really did ask Bill, "What's a budget?" I remembered at Utag, in pre-Mansfield Amendment days, Dave Evans saying to me as hwent off on a trip to ARPA, "We're almost out of money. Got to go get some more." That seemed about right to me. They give you some money. You spend it to find out what to do next. You run out. They give you some more. And so on. PARC never quite made it to that idyllic standard, but for

the first half decade it came close. I needed a group because I had finally ralized that I did not have all of the temperaments required to completely finish an idea. I called it the Learning Research Group (LRG) to be as vaue as possible bout our charter. I only hired people that got stars in their eyes when they heard about the notebook computer idea. I didn't like meetings: didn't believe brainstorming could substitute for cool sustained thought. When anyone asked me what to do, and I didn't have a strong idea, I would point at the notebook nodel and say, "Advance that." LRG members developed a very close relationship with each other--as Dan Ingalls was to say later: "... the rest has enfolded through the love and energy of the whole Learning Research Group." A lot of daytime was spent outside of PARC, playing tennis, bikeriding, drinking beer, eating chinese food, and constantly talking about the Dynabook and its potential to amplify human reach and bring new ways of thinking to a faltering civilization that desperately needed it (that kind of goal was common in California in the afternath of the sixties).

In the summer of '71 I refined the KiddiKomp idea into a tighter design called miniCOM. It used a bit-slice approach like the NOVA 1200, had a bit-map display, a pointing device, a choice of "secondary" (really tertiary) storages, and a language I now called "Smalltalk"--as in "programming should be a matter of ..." and "children should program in ...". The name was also a reaction against the "IndoEuropean god theory" where systems were named Zeus, Odin, and Thor, and hardly did anything. I figured that "Smalltalk" was so innocuous a label that if it ever did anything nice people would be pleasantly surprised.

This Smalltalk language (today labeled -71) was very influenced by FLEX, PLANNER, LOGO, META II, and my own derivatives from them. It was a kind of parser with object-attachment that executed tokens directly. (I think the awkward quoting conventions come from META). I was less interested in programs as algebraic patterns than I was in a clear scheme that could handle a variety of styles of programming. The patterned front-end allowed simple extension, patterns as "data" to be retrieved, a simple way to attach behaviors to objects, and a rudimentary but clear expression of its *eval* in terms that I thought children could understand after a few years experience with simpler programming.. Program storiage was sorted into a discrimintaion net and evalutaion was strightforward pattern-matching.

As I mentioned previously, it was annoying that the surface beauty of LISP was marred by some of its key parts having to be introduced as "special forms" rather than as its supposed universal building block of functions. The actual beauty of LISP

```
Smalltalk-71 Programs

to T 'and' :y do 'y'
to F 'and' :y do F

to 'factorial' 0 is 1
to 'factorial' :n do 'n*factorial n-1'

to 'fact' :n do 'to 'fact' n do factorial n. ^ fact n'

to :e 'is-member-of' [] do F
to :e 'is-member-of' :group
          do'if e = firstof group then T
                          else e is-member-of rest of group'

to 'cons' :x :y is self
to 'hd' ('cons' :a :b) do 'a'
to 'hd' ('cons' :a :b) '<-' :c do 'a <- c'
to 'tl' ('cons' :a :b) do 'b'
to 'tl' ('cons' :a :b) '<-' :c do 'b <- c'
```

came more from the *promise* of its metastrcutures than its actual model. I spent a fair amount of time thinking about how objects could be characterized as universal computers without having to have any exceptions in the central metaphor. What seemed to be needed was complete control over what was passed in a message send; in particular *when* and in *what environment* did expressions get evaluted?

```
to :robot 'pickup' :block
        do 'robot clear-top-of block.
                robot hand move-to block.
                robot hand lift block 50.
                to 'height-of' block do 50'
```

An elegant approach was suggested in a CMU thesis of Dave Fisher [Fisher 70] on the syntheses of control structures. ALGOL60 required a separate link for dynamic subroutine linking and for access to static global state. Fisher showed ow a generalization of these links could be used to simulate a wide variety of control environments. One of the ways to solve the "funarg problem" of LiSP is to associate the proper global tate link with expressions and functions that are to be evaluted later so that the free variables referenced are the ones that were actually implied by the static form of the language. The notion of ";azy evaluation" is anticipated here as well.

Nowadays this approach wouldbe called *reflective design.* Putting it together with the FLEX models suggested that all that should be required for "doing LISP right" or "doing OOP right" would be to handle the mechanics of invocations between modules without having to worry about the details of the modules themselves. The difference between LISP and OOP (or any other system) would then be what the modules could dontain. A universal module (object) referrence --ala B5000 and LISP--and a message holding structure--which could be virutal if the senders and receivers were sympatico-- that could be used by all would do the job.

If all of the fields of a messenger struccture were enumerated according to this view, we would have:

GLOBAL: *the environment of the parameter values*
SENDER: *the sender of the message*
RECEIVER: *the receiver of the message*
REPLY-STYLE: *wiat, fork, ...?*
STATUS: *progress of the message*
REPLY: *eventual result (if any)*
OPERATION SELECTOR: *relative to the receiver*
OF PARAMETERS:
P1:
...:
Pn:

This is a generalization of a stack frame, such as used by the B5000, and very simiilar to what a good intermodule scheme would require in an opeating system such as CAL-TSS--a lot of state for every transactin, but useful to think about.

Much of the pondering during this state of grace (before any workable implementation) had to do with trying to understand what "beautiful" ight mean with reference to object-oriented design. A subjective definition of a beautiful thing is fairly easy but is not of much jelp: we think a thing beautfiul because it evokes certain emotions. The cliche has it like "in the eye of the beholder" so that it is difficult to think of beauty as other than a relation between subject and object in which the predispositions of the subject are all important.

If there are such a thing as universally appealing forms then we can perhaps look to our shared biological heritage for the predispositions. But, for an object like LiSP, it is almost certan that most of the basis of our judgement is leanred and has much to do with other related areas that we think are beautiful, such as much of mathematics.

One part of theperceived beuty of mathematics has to do with a wondrous snery between parasimony, generality, enlightenment, and finesse. For example, the Pythagoriean Theorem is expressable in a single line, is true for all of the infinite number of right triangles, is incredibly uiseful in unerstanding many other relationships, and can be shown be a few simple but profound steps.

When we turn to the various languages for specifying computations we find many to be general and a few to be parsimonious. For example, we can define universal machine languages in just a few instructions that can speicfy anything that can be computed. But most of these we would not call beautiful, in part because the amount and kiind of code tha has to be written to do anything interesting is so contribed and turgid. A simple and small system that can do interesing things also needs a "high slope"--that is a good match between the degree of interestingness and the level of complexity needed to express it.

A fertialized egg that can transform itself into the myriad of specializations needed to make a complex organism has parsimony, gernerality, enlightenment, and finesse-in short, beauty, and a beauty much more in line with my own esthetics. I mean by this that Nature is wonderful both at elegance and practicality--the cell membrane is partly there to allow useful evolutionary kludges to do their neccessary work and still be able act as component by presenting a uniform interface to the world.

One of my continual worries at this time was about the size of the bit-map display. Even if a mixed mode was used (between fine-grained generated characters and coarse-grained general bit-mao for graphics) it would be hard to get enough information on the screen. It occurred to me (ina shower, my favorite place to think) that FLEXtype windows on a bit-map displahy could be made to appear as overlapping documents on a desktop. When an overlapped one was refreshed it would appear to come to the top of the stack. At the time, this did not appear as the wonderful solujtion to the problem but it did have the effect of magnifying the effective area of the display anormously, so I decided to go with it.

To investigate the use of video as a display medium, Bill english and Butler Lampson specified an experimental character generator (built by Roger Bates) for the POLOS (PARC OnLine Office System) terminals. Gary Starkweather had just gotten the first laser printer to work and we ran a coax over to his lab to feed him some text to print. The "SLOT machine" (Scanning Laser Output Terminal) was incredible. The only Xerox copier Gary could get to work on went at 1 page a seocond and could not be slowed down. So Gary just made the laser run at the rate with a resolution of 500 pixels to the inch!

The character generator's font memory turned out to be large enough to simulate a bit-map display f one displayed a fixed "strike" and wrote into the font memory. Ben Laws built a beautiful font editor and he and I spent several months learning about the peculaiarities of the human visual system (it is decidedly non-linear). I was very interested in high-quality text and graphical presentations because I thought it would be easier to get the Dynabook into schools as a "trojan horse" by simply replacing school books rahter than to try to explain to teachers and school boards what was really great about personal computing.

Things were generally going well all over the lab until May of 72 when I tried to get resources to build a few miniCOMs. A relatively new executive ("X") did not want to give them to me. I wrote a memo explaining why the system was a good idea (see Appendix II), and then had a meeting to discuss it. "X" shot it down completely saying amoung other things that we had used too many green stamps getting Xerox to fund the time-shared MAXC and this use of resources for personal machines would confuse them. I was chocked. I crawled away back to the experimental character generator and made a plan to get 4 more made and hooked to NOVAs for the initial kid experiments.

I got Steve Purcell, a summer student from Stanford, to build my design for bit-map painting so the kids could sketch as well as display computer graphics. John Shoch built a line drawing and gesture regognition system (based on Ledeen's [Newman and Sproull 72]) that was integrated with the painting. Bill Duvall of POLOS built a miniNLS that was quite remarkable in its speed and power. The first overlapping windows started to appear. Bob Shur (with Steve Purcell's help) built a 2 1/2 D animation system. Along with Ben Laws' font editor, we could give quite a smashing demo of what we intended to build for real over the next few years. I remember giving one of these to a Xerox executive, including doing a protrait of him in the new painting system, and wound it up with a flourish declaring: "And what's really great about this is that it only has a 20% chance of success. We're taking risk just like you asked us to!" He looked me straigt in the eye and said, "Boy, that's great, but just make sure it works." This was a typical exeuctive notion about risk. He wanted us to be in the "20%" one hundred percent of the time.

That summer while licking my wounds and getting the demo simnulations built and going, Butler Lampson, Peter Deutsch, and I worked out a general scheme for emulated HLL machine languages. I liked the B5000 scheme, but Butler did not want to have to decode bytes, and pointed out that since an 8-bit byte had 256 total possibilities, what we should do is map different meanings onto different parts of the "instruction space." this would give us a "poor man's Huffman code" that would be both flexible and simple. All subsequent emulators at PARC used this general scheme.

I also took another pass at the language for the kids. Jeff Rulifson was a big fan of Piaget (and semiotics) and we had many discussions about the "stages" and what iconic thinking might be about. After reading Piaget and especially Jerome Bruner, I was worried that the directly symbolic approach taken by FLEX , LOGO (and the current Smalltalk) would be difficult for the kids to process since evidence existed that the symbolic stage (or mentality) was just starting to switch on. In fact, all of the educators that I admired (including Montessori, Holt, and Suzuki) all seemed to call for a more figurative, more iconic approach. Rduolph Arnheim [Arnheim 69] had written a classic book about visual thinking, and so had the eminent art critic Gomrich [Gombrich **]. It really seemed that something better needed to be done here. GRAIL wasn't it, because its use of imagery was to portray and edit flowcharts, which seemed like a great step backwards. But Rovner's AMBIT-G held considerably more promise [Rovner 68]. It was kind of a visual SNOBOL [Farber 63]

and the pattern matching ideas looked like they would work for the more PLANNERlike scheme I was using.

Bill English was still encouraging me to do more reasonable appearing things to get higher credibility, likemakin budgets, writing plans and milestone notes, so I wrote a plan that proposed over the next few years that we would build a real system on the character generators cum NOVAs that would involve OOP, windows, painting, music, animation, and "iconic programming." The latter was deemed to be hard and would be handled by the usual method for hard problems, namely, give them to grad students.

IV. 1972-76--The first real Smalltalk (-72), its birth, applications, and improvements

In Sept. within a few weeks of each other, two bets happened that changed most of my plans. First, Butler and Chuck came over and asked: "Do you have any money?" I said, "Yes, about $230K for NOVAs and CGs. Why?" They said, "How would you like us to build your little machine for you?" I said, "I'd like it fine. What is it?" Butler said: "I want a '$500 PDP-10', Chuck wants a '10 times faster NOVA', and you want a 'kiddicomp'. What do you need on it?" I told them most of the results we had gotten from the fonts, painting, resolution, animation, and music studies. I aksed where this had come from all of a sudden and Butler told me that they wanted to do it anyway, that Executive "X" was away for a few months on a "task force" so maybe they could "Sneak it in", and that Chuck had a bet with Bill Vitic that he could do a whole machine in just 3 months. "Oh," I said.

The second bet had even more surprising results. I had expected that the new Smalltalk would be an iconic language and would take at least two years to invent, but fate intervened. One day, in a typical PARC hallway bullsession, Ted Kaeh;er, Dan Ingalls, and I were standing around talking about programming languages. The subject pf power came up and the two of them wondered how large a language one would have to make to get great power. With as much panache as I could muster, I asserted that you could define the "most powerful language in the world" in "a page of code." They said, "Put up or shut up."

Ted went back to CMU but Can was still around egging me on. For the next two weeks I got to PARC every morning at four o'clock and worked on the problem until eight, when Dan, joined by Henry Fuchs, John Shoch, and Steve Prcell shoed up to kibbitz the mroning's work.

I had orignally made the boast because McCarthy's self-describing LISP interpreter was written in itself. It was about "a page", and as far as power goes, LISP was the whole nine-yards for functional languages. I was quite sure I could do the same for object-oriented languages *plus* be able to do a resonable syntax for the code *a loa* some of the FLEX machine techiques.

It turned out to be more difficult than I had first thought for three reasons. First, I wanted the program to be more like McCarthy's second non-recursive interpreter--the one implemented as a loop that tried to resemble the original 709 implementation of Steve Russell as much as possible. It was more "real". Second, the intertwining of the "parsing" with message receipt--the evaluation of paremters which was handled separately in LISP--required that my object-oriented interpreter re-enter itself "sooner" (in fact, much sooner)

than LISP required. And, finally, I was still not clear how *send* and *receive* should work with each other.

The first few versions had flaws tha wee soundly criticized by the group. But by morning 8 or so, a version appeared that seemed to work (see Appendix III for a sketch of how the interpreter was designed). The major differences from the official Smalltalk-72 of a little bit later were that in the first version symbols were byte-coded and the reeiving of return of return-values from a send was symmetric--i.e. reciept could be like parameter binding--this was particular useful for the return of multiple values. for various reasons, this was abandoned in favor of a more expression-oriented functional return style.

Of course, I had gone to considerable pains to avoid doing any "real work" for the bet, but I felt I had proved my point. This had been an interesting holiday from our official "iconic programming" pursuits, and I thought that would be the end of it. Much to my surprise, ionly a few ays later, Dan Ingalls shoed me the scheme working on the NOVA. He had coded it up (in BASIC!), added a lot of details, such as a token scanner, a list maker, etc., and there it was--running. As he liked to say: "You just do it and it's done."

It evaluted `3 = 4` very slowly (it was "glacial", as Butler liked to say) but the answer alwas came out 7. Well, there was nothing to do but keep going. Dan loved to bootstrap on a system that "always ran," and over the next ten years he made at least 80 major releases of various flavors of Smalltalk.

In November, I presented these ideas and a demonstration of the interpretation scheme to the MIT AI lab. This eventuall led to Carl Hewitt's more formal "Actor" approach [Hewitt 73]. In the first Actor paper the resemblence to Smalltalk is at its closest. The paths later diverged, partly because we were much more interested in making things than theorizing, and partly because we had something no one else had: Chuck Thacker's Interim Dynabook (later known as the "ALTO").

Just before Check started work on the machine I gave a paper to the National Council of Teachers of English [Kay 72c] on the Dynabook and its potential as a learning and thinking amplifier--the paper was an extensive rotogravure of "20 things to do with a Dynabook" [Kay 72c]. By the time I got back from Minnesota, Stewart Brand's Rolling Stone article about PARC [Brand 1972] and the surrounding hacker community had hit the stands. To our enormous surprise it caused a major furor at Xerox headquarters in STamford, Connectitcut. Though it was a wonderful article that really caught the spirit of the whole culture, Xerox went berserk, forced us to wear badges (over the years many were printed on t-shirts), and severely restricted the kinds of publications that could be made. This was particular disastrous for LRG, since we were the "lunatic fringe" (so-called by the other computer scientists), were planning to go out to the schools, and needed to share our ideas (and programs) with our colleagues such as Seymour Papert and Don Norman.

Executive "X" apparently heard some harsh words at Stamford about us, because when he returned around Christmas and found out about the interim Dynabook, he got even more angry and tried to kill it. Butler wound up writing a masterful defence of the machine to hold him off, and he went back to his "task force."

Check had started his "bet" on November 22, 1972. He and two tecnicians did all of the machine except for the disk interface which was done by Ed McCreight. It had a ~500,000

pixel (606x808) bitmap display, its microcode instruction rate was about 6 MIPS, it had a grand total of 128k, and the entire machine (exclusive of the memory) ws rendered in 160 MSI chips distributed on two cards. It ws eautiful [Thacker 1972, 1986]. One of the wonderful features of the machine was "zero-over-head" tasking. It had 16 program counters, one for each task. Condition falgs were tied to interesting events (such as "horizontal retrace pulse", and "disk sector pulse", etc.). Lookaside logic scanned the flags while the current instruction was executing and picked the highes prioritity program counter to fetch from next. The machine never had to wait, and the reuslt was that most hardware functions (particularly those that involved i/o (like feeding the display and handling the disk) could be replaced by microcode. Even the refresh of the MOS dynamic RAM was done by a task. In other words, this was a coroutine architecture. Check claimed that he got the idea from a lecture I had given on coroutines a few months before, but I remembered that Wes Clark's TX-2 (the Sketchpad machine) had used the idea first, and I probably mentioned that in the talk.

In early April, just a little over three months fromthe start, the first Interim Dynabook, known as 'Bilbo,' greeted the world and we had the first bit-map picture on the screen within minutes; the Muppets' Cookie Monster that I had sketched on our painting system.

Soon Dan had bootstrapped Smalltalk across , and for many months it was the sole software sytem to run on the Interim dynabook. Appendix I has an "acknowledgements" dodcument I wrote from this time that is interesting it its allocation of credits and the various priorities associated with them. My $230K was enough to get 15 of the original projected 30 machines (over the years some 2000 Interim Dynabooks were actually built. True to Schopenhauer's observation, Executive "X" now decided that the Interim Dynabook was a good idea and he wanted all but two for his lab (I was in the other lab). I had to go to considerable lengths to get our machines back, but finally succeeded.

By this time most of Smalltalk's schemes had been sorted out into six main ideas that were in accord with the initial premises in designing the interpreter. The ist three principales are what objects "are about"--how they are seen and used from "the outside." Thse did not require any modification over the years. The last three --objects from the inside--were tinkered with in every version of Smalltalk (and in subsequent OOP designs). In this scheme (1 & 4) imply that classes are objects and that they must be instances of themself. (6) implies a LiSPlike universal syntax, but with the reeiving object as the first item followed by the message. Thus c_i <- de (with subscripting rendered as "o" and multiplication as "*") means:

receiver message
c *o i <- d*e*

The c is bound to the receiving object, and all of o i <- d*e is the message to it. The message is made up of literal token ".", an expression to be evaluated in the sender's context (in this case i), another literal token <-, followed by an expression to be evaluated in the sender's context (d*e). Since "LISP" paris are made from 2 element objets they can be indexed more simply: c

1. Everything is an object
2. Objects communicate by sending and receiving *messages* (in terms of objects)
3. Objects have their *own memory* (in terms of objects)
4. Every object is an instance of a *class* (which must be an object)
5. The class holds the shared *behavior* for its instances (in the form of objects in a pogram list
6. *To eval a program list, control is*

hd, c tl, and c hd <- foo, etc.

"Simple" expressions like *a+b* and *3+4* seemed more troublesome at first. Did it really make sense to think of them as:

receiver	message
a	*+ b*
3	*+ 4*

passed to the first object and the remainder is treated as its message

It seemed silly if only integers were considered, but there are many other metaphoric readings of "+", such as:

"kitty" + "kat" => "kittykat"
[3 4 5 6 7 8] + 4 => [7 8 9 10 11 12]

This led to a style of finding *generic behaviors* for message symbols. "Polymorphism" is the official term (I believe derived from Strachey), but it is not really apt as its original meaning applied only to functions that could take more than one type of argument. An example class of objects in Smalltalk-72, such as a model of CONS pairs, would look like:

Since control is passed to the class before any of the rest of the message is considered--the class can decide not to receive at its discretion--complete protection is retained. Smalltalk-72 objects are "shiny" and impervious to attack. Part of the environment is the binding of the SENDER in the "messenger object" (a generalized activation record) which allows the receiver to determine differential provileges (see Appendix II for more details). This looked ahead to the eventualuse of Smalltalk as a network OS (See [Goldstein & Bobrow 1980]), and I don't recall it being used very much in Smalltalk-72.

One of the styles retained from Smalltalk-71 was the comingling of function and class ideas. In other works, Smalltalk-72 classes looked like and could be used as functions, but it was easy to produce an instance (a kind of closure) by using the object ISNEW. Thus factorial could be written "extensionally" as:

*to fact n (^if :n=0 then 1 else n*fact n-1)*

or "intensionally," as part of class integer:

*(... o! * (^:n=1) * (1) (n-1)!)*

Of course, the whole idea of Smalltalk (and OOP in general) is to define everything *intensionally*

Proposed Smalltalk-72 Syntax

```
Pair :h :t
    hd <- :h
        hd              = h
        tl <- :t
        tl              = t
        isPair          = true
        print           =  '( print. SELF mprint.
```

```
mprint          = h print. if t isNil then ') print
                  else if t isPair then t mprint
                  else '* print. t print. ') print
length          = 1 + if t isList then t length else 0
```

And this was the direction of movement as we learned how to progam in the new style. I never liked this syntax (too many parentheses and nestings) and wanted something flatter and more grammar-like as in Smalltalk-71. To the right is an example syntax from the notes of a talk I gave around then. We will see something more like this a few years later in Dan's design for Smalltalk-76. I think something simlar happened with LISP--that the "reality" of the straightforward and practical syntax you could program in prevailed against the flights of fancy that never quite got built.

Development of the Smalltalk-72 System and Applications

The advent of a real Smalltalk on a real machine started off an explosion of parallel paths that are too difficult to intertwine in strict historical order. Let me first present the genera development of the Smalltalk-72 system up to the transistion to Smalltalk-76, and then follow that with the several years of work with children that were the primary mnotivation for the prroject. The Smalltalk-72 interpreter on the Interim Dynabook was not exactly a zippy ("majestic" was Butler's pronouncement), but was easy to change and quite fast enough for many real-time interactive systems to be built in it.

Overlapping windows were the first project tackled (With Diana Merry) after writing the code to read the keyboard and create a string of text. Diana built an early version of a bit field block transfer (bitblt) for displaying variable pitch fonts and generally writing on the display. The first window versions were done as real 2 1/2 D draggable objects that were just a little too slow to be useful. We decided to wait until Steve Purcell got his animation system going to do it right, and opted for the style that is still in use today, which is more like a "2 1/4 D". Windows were perhaps the most redesigned and reimolemented class in Smalltalk because we didn't quite have enough compute power to just do the continual viewing to "world coordinates" and refereshgin that my former Utag colleagues were starting to experiment with on the flight simulator projects at Evans & Sutherland. This is a simple powerful model but it is difficult to do in real-time even in 2 1/2D. The first practical windows in Smalltalk used the GRAIL conventions of sensitive corners for moving, resizing, cloning, and closing. Window scheduling used a simple "loopless" control scheme that threaded all of the windows together.

One of the next classes to be implemented on the Interim Dynabook (after the basics of numbers, strings, etc.) was an object-oriented version of the LOGO turtle implemented by Ted. This could make many turtle instances that were used both for drawing and as a kind of value for graphics transformations. Dan created a class of "commander" turtles that could control a troop of turtles. Soon the tutles were made so they could be clipped by the windows.

John Shoch built a mouse-driven structured editor for Smalltalk code.

Larry Tesler (then working for POLOS) did not like the modiness and general appraoch of NLS, and he wanted both show the former NLSers an alternative and to conduct some user studies (almost unherard of in those days) about editing. This led to his programming miniMOUSE in Smalltalk, the first real WYSIWYG galley editor at PARC. It was modeless

(almost) and fun to use, not just for us but for the many people he tested it on (I ran the camera for the movies we took and remember their delight and enjoyment). miniMOUSE quickly became an alternate editor for Smalltalk code and some of the best demos we ever gave used it.

One of the "small program" projects I tried on an adult class in the Spring of '74 was a one-page paragraph editor. It turned out to be too complicated, but the example I did to show them was completely modeless (it was in the air) and became the basis for much of the Smalltalk text work over the next few years. Most of the improvements were mde by Dan and Diana Merry. Of course, objects mean multi-media documents, you almost get them for free. Early on we realised that in such a document, each component object should handle its own editing chores. Steve Weyer built some of the earliest multi-media documents, whose range was greatly and variously expanded over the years by Bob Flegal, Diana Merry, Larry Tesler, Tim Mott, and Trygve Reenskaug.

Steve Weyer and I devised *Findit*, a "retrival by example" interface that used the analogy of classes to their instances to form retrieval requests. This was used for many years by the PARC library to control ciruculation.

The sampling synthese music I had developed on the NOVA col generate 3 high-quality real-time voices. Bob Shur and Chuck Thacker transfered the scheme to the Interim Dynabook and achieved 12 voices in real-time. The 256 bit generalized input that we had specified for low speed devices (used for the mouse and keyboard) made it easy to connect 154 more to wire up two organ keyboards and a pedal. Effects such as portamento and decay were programmed. Ted Kaehler wrote TWANG, a music capture and editing system, using a tablature notation that we devised to make music clear to children [Kay 1977a]. One of the things that was hard to do with sampling was the voltage controlled operator (VCO) effects that were popular on the "Well Tempered Synthesizer." A symmer later, Steve Saunders, another of our bright summer students, was challenged to find a way to accomplish John Chowning's very non-real-time FM synthesis in real-time on the ID. He had to find a completely different way to think of it than "FM", and succeeded brilliantly with 8 real-time voices that were integrated into TWANG [Saunders *].

Chris Jeffers (who was a musician and educator, not a computer scientist) knocked us out with OPUS, the first real-time score capturing system. Unlike most systems today it did not require metronomic playing but instead took a first pass lookng for string and weak beats (the phrasing) to establish a local model of the likely temp fluctuations and then used curve fitting and extrapolation to make judgements about just where in the measure, and for what time value, a given note had been struck.

The animmations on the NOVA ran 3-5 objects at about 2-3 frames per second. Fast enough for the *phi* phenomenon to work (if double buffering was used), but we wanted "Disney rates" of 10-15 frames a second for 10 or more large objects and many more smaller ones. This task was put into the engenious hands of Steve Purcell. By the fall of '73 he could demo 80 ping-pong balls and 10 flying horses running at 10 frames per second in 2 1/2D. His next task was to make the demo into a general systems facility from which we could construct animation systems. His CHAOS system started working in May '74, just in time for summer visitors Ron Baecker, Tom Horseeley, and professional animator Eric Martin to visit and build SHAZAM a marvelously capable and simple animation system based on Ron's GENESYS thesis project on the TX-2 in the late sixties [Baecker 69].

The main thesis project during this time was DAve Smith's PYGMALION [Smith 75], an essay into iconic programming (no, we hadn't quite forgotton). One progammed by showing the system how hanges should be made, much as one would illustrate on a balackboard with another programmer. This programm became the starting place from which many subsequent programming by example" systems took off.

I should say something about the size of these programs. PYGMALION was the largest program ever written in Smalltalk-72. It was about 20 pages of code--all that would fit in the interim dynabook ALTO--and is given in full in Smith s thesis. All of the other applications were smaller. For example, the SHAZAM animation system was written and revised several times in the summer of 1974, and finally woulnd up as a 5-6 page application which included its icon-controlled multiwindowed user interface.

Given its roots in simulation languages, it was easy to write in a few pages Simpula, a simple version of the SiMULA sequencing set approach to scheduling. By this time we had decided that coroutines could be more cleanly be rendered by scheduling individual methods as separate simulation phases. The generic SIMULA example was a job shop. This could be genearlized into many useful forms such as a hospital with departments of resources serving patients (see to the right). The children did not care for hosipitals but saw th they could model amusement parks, like Disneyland, their schools, the stores they and their parents shopped in, and so forth. Later this model formed the basis of the Smalltalk Sim-Kit, a high-level end-user programming enviornment (described ahead).

Many nice "computer sciency" constructs were easy to make in Smalltalk-72. For example, one of the controversies of the day was whether to have gotos or not (we didn't), and if not, how could certain very useful control strcutres--such as multiple exits from a loop--be specified? Chuck Zahn at

```
(until Return or Delete do
    ('character <- display <- keyboard.
    character = ret > (Return)
    character = del > (Delete)
    )
then case
    Return: ('deal with this normal exit')
    Delete: ('handle the abnormal exit'))
```

SLAC proposed an event-driven case structure in which a set of events could be defined so that when an event is encountered, the loop will be exited and the event will select a statement in a cas block [Zahn 1974, Knuth 1974]. Suppose we want to write a simple loop that reads characters from the keyboard and outputs them to a display. We want it to exit normally when the <return> key is struck and with an error if the <delete> key is hit. Appendix IV shows how John Shoch defined this control structure.

The Evolution of Smalltalk-72

Smalltalk-74 (sometimes known as FastTalk) was a version of Smalltalk-72 incorporating major improvements which included providing a real "messenger" object, message dictionaries for classes (a step towards real class objects), Diana Merry's bitblt (the now famous 2D graphics operator for bitmap graphics) redesigned by Dan and implmented in microcode, and a better, more general window interface. Dave Robson while a student at UC Irvine ha dheard of our project and made a pretty good stab at implementeing an OOPL. We invited him for a summer and never let him go back--he was a great help in formulating an official semantics for Smalltalk.

The crowning addition was the OOZE (Object Oriented Zoned Environment) virtual memory system tat served Samlltalk-74, and more importantly, Smalltalk-76 [Ing 78, Kae *]. The ALTO was not ver large (128-256K), especially with its page0sized display (64k), and even with small programs, we soon ran out of storage. The 2.4 megabyte model 30 desk drive was faster and larger than a floppy and slower and smaller than today's hard drives. It was quite similar to the HP direct contact disk of the FLEX machine on which I had tried a fine-grain version of the B5000 segment swapper. It had not worked as well as I wanted, despite a few good ideas as to how to choose objects when purging. When the gang wanted to adopt this baic scheme, I said: "But I never got it to work well." I remember Ted Kaehler saying, "Dont' worry, we'll make it work!"

The basic idea in all of these systems is to be able to gather the most comprehensive possible working set of objects. This is most easily accomplished by swapping individual objects. Now the problem becomes the overhead of purging non-working set objects to make room for the ones that are needed. (Paging sometimes works better for this part because you can get more than on object (OOZE) in each disk touch.) Two ideas help a lot. First, Butler's insight in the GENTE OS that it was worthwhile to expend a small percentage of tie purging dirty objects to make core as clean as possible [Lampson 1966]. Thus crashes tend not to hurt as much and there is alwasy clean storage to fetch pages or objects from the disk into. The other is one from the FLEX system in which I set up a stochastic decision mechanism (based on the class of an object) that determined during a purge whether or not to throw an object out. This had two benefits: important objects tended not to go out, and a mistake would just bring it back in again with the distribution insuring a low probability that the object would be purged again soon.

The other problem that had to be taken care of was object-pointer integrity (and this is where I had failed in the FLEX machine to come up with a good enough solution). Wht was needed really was a complete transaction, a brand new technique (thought up by Butler?) that ensured recover regardless of when the system crashed. This was called "cosmic ray protection" as the early ALTOS had a way of just crashing once or twice a day for no discernable good reason. This, by the way did not particularly bother anyone as it was fairly easy to come up with undo anbd replay mechanisms to get around the cosmic rays. For pointer-based systems that had automatic storage management, this was a bit more tricky.

Ted and Dan decided to control storage using a Resident Object Table that was the only place machine addresses for objects would be found. Other useful information was stashed there as well to help LRU aging. Purging was done in background by picking a class, positioning the disk to its instances (all of a particular class were stored together), then running through the ROT to find the dirty ones in storage and stream them out. This was pretty efficient and, true to Butler's insight, furnished a good sized pool of clean storage that could be overwritten. The key to the design though (and the implementation of the transaction mechanism) was the checkpointing scheme they came up with. This insured that there was a recoverable image no more than a few seconds old, regardless of when a crash ight occur. OOZE swapped objects in just 80kb of working storage and could handle about 65K objects (up to several megabytes worth, more than enough for the entire system, its interface, and its applications).

"Object-oriented" Style

This is probably a good place to comment on the difference between what we thought of as

OOP-style and the superficial encapsulation called "abstact data types" that was just starting to be investigated in academic circles. Our early "LISP-pair" definition is an example of an abstract data type because it preserves the "field access" and "field rebinding" that is the hallmark of a data structure. Considerable work in the 60s was concerned with generalizing such strcutures [DSP *]. The "official" computer science world started to regard Simula as a possible vehicle for defining abstract data types (even by one of its inventors [Dahl 1970]), and it formed much of the later backbone of ADA. This led to the ubiquitous stack data-type example in hundreds of papers. To put it mildly, we were quite amazed at this, since to us, what Simula had whispered was something much stringer than simply reimplementing a weak and ad hoc idea. What I got from Simula was that you could now replace bindings and assignment with *goals*. The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as *site of higher level behaviors more appropriate for use as dynamic components*.

Even the way we taught children (cf. ahead) reflected this way of looking at objects. Not too surprisingly this approach has considerable bearing on the ease of programming, the size of the code needed, the integrity of the design, etc. It is unfortunate that much of what is called "object-oriented programming" today is simply old style programming with fancier constructs. Many programs are loaded with "assignment-style" operations now done by more expensive attached procedures.

Where does the special efficiency of object-oriented design come from? This is a good question given that it can be viewed as a slightly different way to apply procedures to data-structures. Part of the effect comes from a much clearer way to represent a complex system. Here, the constraints are as useful as the generalities. Four techniques used together--persistent state, polymorphism, instantiation, and methods-as-goals for the object--account for much of the power. None of these require an "object-oriented language" to be employed--ALGOL 68 can almost be turned to this style--and OOPL merely focuses the designer's mind in a particular fruitful direction. However, doing encapsulation right is a commitment not just to abstraction of state, but to eliminate state oriented metaphors from programming.

Perhaps the most important principle--again derived from operating system architectures--is that when you give someone a structure, rarely do you want them to have unlimited priviledges with it. Just doing type-matching isn't even close to what's needed. Nor is it terribly useful to have some objects protected and others not. Make them all first class citizens and protect all.

I believe that the much smaller size of a good OOP system comes not just by being gently forced to come up with a more thought out design. I think it also has to do with the "bang per line of code" you can get with OOP. The object carries with it a lot of significance and intention, its methods suggest the strongest kinds of goals it can carry out, its superclasses can add up to much more code-frunctionality being invoked than most procedures-on-data-structures. Assignment statements--even abstract ones--express very low-level goals, and more of them will be needed to get anything done. Generally, we don't want the programmer to be messing around with state, whether simulated or not. The ability to instantiate an object has a considerable effect on code size as well. Another way to think of all this is: though the late-binding of automatic storage allocations doesn't do anything a programmer can't do, its presence leads both to simpler and more powerful code. OOP is a late binding strategy for many things and all of them together hold off fragility and size explosion much longer than the older methodologies. In other words, human programmers

aren't Turing machines--and the lesss their programming systems require Turing machine techniques the better.

Smalltalk and Children

Now that I have summarized the "adult" activities (we were actually only semiadults) in Smalltalk up to 1976, let me return to the summer of '73, when we were ready to start experiments with children. None of us knew anything about working with children, but we knew that Adele Goldberg and Steve Weyer who were then with Pat Suppes at Standford had done quite a bit and we were able to entice them to join us.

Since we had no idea how to teach object-oriented programming to children (or anyone else), the first experiments Adele did mimicked LOGO turtle graphics, and she got what appeared to be very similar results. That is to say, the children could get the turtle to draw pictures on the screen, but there seemed to be little happening beyond surface effects. At that time I felt that since the content of personal computering was interactive tools, that the content of this new kind of authoring literacy should be the creation of interactive *tools* by the children. Procedural turtle graphics just wasn't it.

The Adele came up with a breillian approach to teaching Smalltalk as an object-oriented language: the "Joe Book." I believe this was partly influneced by Minsky's idea that you should teach a programming language holistically from working examples of serious programs.

Several instances of the class box are created and sent messages, culminating with a simple multiprocess animation. After getting kids to guess what a box might be like--they could come surprisingly close--they would be shown:

```
to box | x y size tilt
(odraw   =    (@place x y turn tilt. square size.
oundraw  =    (@ white, SELF draw, @black)
oturn    =    (SELF undraw. 'tilt <- tilt + :. SELF draw)
ogrow    =    (SELF undraw. 'size <- size + :. SELF draw)
ISNEW    =    (SELF undraw. 'size <- size + :. SELF draw)
```

What was so wonderful about this idea were the myriad of children's projects that could spring off the humble boxes. And some of the earliest were tools! This was when we got really excited. For example, Marion Goldeen's (12 yrs old) painting system was a full-fledged tool. A few yuears later, so was Susan Hamet's (12 yrs old) OOP illustration system (with a design that was like the MacDraw to come). Two more were Bruce Horn's (15 yrs old) music score capture system and Steve Ptz's (15 yrs old) circuit design system. Looking back, this could be called another example in computer science of the "early success syndrome." The successes were real, but they weren't as general as we thought. They wouldn't extend into the future as stringly as we hoped. The children were chosen from the Palo Alto schools (hardly an average background) and we tended to be much more excited about the successes than the difficulties. In part, that we were seeing was the "hack phenomenon," that, for any given pursuit, a particular 5% of the population will jump into it naturally, while the 80% or so who can learn it in time do not find it at all natural.

We had a dim sense of this, but we kept on having relative successes. We could definitely see that learning the mechanics of the system was not a major problem the children could

get mose of it themsleves by swarming over the ALTOS with Adele's JOE book. The problem seemed more to be that of design.

It started to hit home in the Spring of '74 after I taught Smalltalk to 20 PARC nonprogrammer adults. They were able to get through the initial material faster than the children, but just as it looked like an overwhelming success was at hand, they started to crash on problems that didn't look to me to be much harder than the ones they had just been doing well on. One of them was a project thought up by one of the adults, which was to make a little database system that could act like a card file or rolodex. They couldn't even come close to programming it. I was very surprised because I "klnew" that such a project was well below the mythical "two pages" for end-users we were working within. That night I worote it out, and the next day I showed all of them how to do it. Still, none of them were able to do it by themsleves. Later, I sat in the room poindering the board from my talk. Finally, I counted the number of nonobvious ideas in this little program. They came to 17. And some of them were like the concept of the arch in building design: very hard to discover, if you don't already know them.

The connection to literacy was painfully clear. It isn't enough to just learn to read and write. There is also a literature that renders ideas. Language is used to read and write about them, but at some point the organization of ideas starts to dominate mre language abilities. And it help greatly to have some powerful ideas under one's belt to better acquire more powerful ideas [Papert 70s]. So, we decided we should teach *design*. And Adele came up with another brillian stroke to deal with this. She decided that what was needed was in intermediary between the vague ideas about the problem and the very detailed writing and debugging that had to be done to get it to run in Smalltalk. She called the intermediary forms *design templates*.

Using these the children could look at a situation they wanted to simulate, and decompose it into classes and messages without having to worry just how a method would work. The method planning could then be done informally in English, and these notes would later serve as commentaries and guides to the writing of the actual code. This was a terrific idea, and it worked very well.

But not enough to satisfy us. As Adele liked to point out, it is hard to claim success if only some of the children are successful--and if a mximum effort of both children and teachers was required to get the successes to happen. REal pedagogy has to work in much less idealistic settings and be consderably more robust. Still, some successes are qualitatively different from no successes. We wanted more, and started to push on the inheritence idea as a way to let novices build on frameworks that could only be designed by experts. We had good reason to believe that this could work because we had been impressed by Lisa vanStone's ability to make significant changes to SHAZAM (the fix or six page Smalltalk animation tool done by relatively expert adults). Unfortunately, inerhitence--though an incredibly powerful technique--has turned out to be very difficult for novices (and even professionals) to deal with.

AT this point, let me do a look back from the vantage point of today. I'm now pretty much convinced that our design template approach was a good one after all. We just didn't apply it longitudinally enough. I mean by this that there is now a large accumulation of results from many attempts to teach novices programming [Soloway 1989]. They all have similar stories that seem to have little to do with the various features of the programming languages used, and everything to do with the difficulties novices have thinking the special way that

good programmers think. Even with a much better interface than we had then (and have today), it is likely that this rea is actually more like writing than we wanted it to be. Namely, for the "80%", it really has to be learned gradually over a period of years in order to build up the structures that need to be there for design and solution look-ahead.[41]

The problem is not to get the kids to do stuff--they love to do, even when they are not sure exactly what they are doing. This correlates well with studies of early learning of language, when much rehearsal is done regardless of whether content is involved. Just *doing* seems to help. What is difficult is to detrmine *what* ideas to put forth and how *deeply* they should penetrate at a given child's developmental level. This is a confusion still persists for reading and writing of natural language--and for mathematics--despite centuries of experience. And it is the main hurdle for teaching children programming. When, in what order and depth, and how should the powerful ideas be taught?

Should we even try to teach programming? I have met hundreds of programmers in the last 30 years and can see no discernable influence of programming on their general abiltity to think well or to take an enlightened stance on human knowledge. If anything, the opposite is true. Expert knowledge often remains rooted in the environments in which it was first learned--and most metaphorical extensions result in misleading analogies. A remarkable number off artists, scientists, philosophers are quite dull outside of their specialty (and one suspects within it as well). The first siren's song we need to be wary of is the one that promises a connection between an interesting pursuit and interesting thoughts. The music is not in the piano, and it is possible to graduate Julliard wiothout finding or feeling it.

I have also met a few people for whom computing provides an important new mataphor for thinkingg about human knowledge and reach. But something else was needed besides computing for enlightenment to happen.

Tools provide a path, a context, and almost an excuse for developing enlightenment, but no tool ever contained it or can dispense it. Ceasare Pavese observed: to know the world we must construct it. In other words, we make not just to have, but to know. but the having can happen without most of the knowing taking place.

Another way to look at this is that knowledge is in it s least ineresting state when it is first being learned. the representations--whether marking, allutions, or phisical control--get in the way (almost take over as goals) and must be laboriously and painfully interpreted. From here there are several useful paths, two of which are important and intertwined.

The first is *fluency*, which in part is the process of building mental structures that disappear the interpretation of the representations. The letters and words of a sentence are experienced as meaning rather than markings, the tennis reaquet or keyuboard becomes an extension of one's body, and so forth. If carried further one eventually becomes a kind of expert--but without deep knwledge in other areas, attenpts to generalize are usually too crisp and ill formed.

The second path is towards taking the knowledge as a *metaphor* than can illuminate other areas. But without fluency it is more likely that prior knowledge will hld sway and the matphors from this side will be fuzzy and misleading.

The "trick," and I think that this is waht liberal arts educations is supposed to be about, is to

get fluent and deep while building realtionships with other fluent deep knowledge. Our society has lowered its aims so far that it is happy with "increases in scores" without daring to inquire whether any important threshold has been crossed. Being able to read a warning on a pill bottle or write about a summer vacation is not literacy and our society should nbot treat it so. Literacy, for example is being able to fluently read and follow the 50 oage argument in Paine's Common Sense and being able (and happy) to fluently write a critique or defence of it. Another kind of 20th century literacy is being able to hear about a new fatal contagious incurable disease and instantly know that a disastrous exponential relationship holds and early action is of the highest priority. Another kind of literacy would take citizens to their personal computers where they can fluently and without pain build a systems simulation of the disease to use as a comparison against further information.

At the liberal arts level we would expect that connections between each of the fluencies would form truly powerful metaphors for considering ideas on the light of others.

The reason, therefore, that many of us want children to understand computing deeply and fluently is that like literature, matematics, science, music, and art, it carries special ways of thinking about situations that in contra=st with other knowledge and other wasy of thinking critically boost our ability to understand our world.

We did not know then, and I'm sorry to say from 15 years later, that these critical questions still do not yet have really useful answers. But there are some indications. Even very young children can understand and use interactive *transformational* tools. The first ones are their hands! The can readily extend these experiences to cimputer objects and making changes to them. They can often imagine what a proposed change will do and not be surprised at the result. Two and three year olds can use the Smalltalk-style interface and manipulate object-oriented graphics. Third graders can (in a few days) learn more than 50 features--most of these are transformational tools--of a new system includeing its user interface. They can answer any question whose answer requires the application of just one of these tools. But it is extremely difficult for them to answer any question that requires two or more transformations. Yet they have no problem applying sequenes of transformations, exploring "forward." It is for conceiving and achieving even modest goals requiring several changes that they almost completly lack navigation abilities.

It seems that what needs to be learned and taught is now to package up transformations in twos and threes in a manner similar to learning a strategic game like checkers. The vague sense of a "threesome" pointing twoards one's goal can be a set up for the more detailed work that is needed to accomplish it. This art is possible for a large percentage of the population, but for most, it will need to be learned gradually over several years.

V. 1976-80--The first modern Smalltalk (-76), its birth, applications, and improvements

By the end of 1975 I felt that we were losing our balance--that the "Dynabook for children" idea was slowily dimming out--or perhaps starting to be overwhelmed by professional needs. In January 1976, I took the whole group to Pajaro Dunes for a three day offsite to bring up the issues and try to reset the compass. It was called "Let's Burn Our Disk Packs." There were no shouting matches, the group liked (I would go so far to say: loved) each other too much for that. But we were troubled. I used the old aphorism that "no biological organism can live in its own waste products" to please for a *really* fresh start: a hw-sw

system very different from the ALTO and Smalltalk, One thing we all did agree on was that the current Smalltalk's power did not match our various levels of aspiration. I thought we needed something different, as I did not see how OOP by itself was going to solve our end-user problems. Others, particularly some of the grad students, really wanted a better Smalltalk that was faster and could be used for bigger problems. I think Dan felt that a better Smalltalk could be the vehicle for the different system I wanted, but could not describe clearly. The meeting was not a disaster, and we went back to PARC still friends and colleagues, but the absolute cohesiveness of the first four years never rejelled. I started designing a new small machine and language I called the *NoteTaker* and dan started to design Smalltalk-76.

The reason I wanted to "burn the disk packs" is that I had a very McLuhanish feeling about media and environments: that once we've shaped tools, in his words, they hum around and "reshape us." Of course this is a great idea if the tools are really good and aimed squarey at the issues in question. But the other edge of the sword cuts as deep--that inadquate tools and environments *still* reshape our thinking in spite of their problems, in part, because we want paradigms to guide our goals. Strong paradigms like LISP and Smalltalk are so compelling that they eat their young: when you look at an application in either of these two systems, they resemble the systems themselves, not a new idea. When I looked at Smalltalk in 1975, I was looking at something great, but I did not see an enduser language, I did not see a solution to the original goal of a "reading" and "writing" computer medium for children. I wanted to stop, dynamite everything and start from scratch again.

The *NoteTaker* was to be a "laptop" that could be built in a few years using the (almost) available 16K RAMS (a vast improvement over the 1K RAMS that the ALTO employed). A laptop couldn't use a mouse (which I hated anyway) and a table seemed awkward (not a lot of room and the stylus could flop out of reach when let go), so I came up with an embedded pointing device I called a "tabmouse." It was a relative pointer and had an *up* sensor so it could be stroked like a mouse and would also stay where you left it, but it felt like a stylus and used a pantograph mechanism that eliminated the annoying hysteresis bias in the x and y directions that made it hard to use a mouse as a pen. I planned to use a multiprocessor architecture of slow but highly integrated chips as originally specified for the Dynabook and wanted a new bytecoded interpreter for a friendlier and simpler system than Smalltalk-72.

Meanwhile Dan was proceeding with his total revamp of Smalltalk and along somewhat similar lines [In 78]. The first major thing that needed to be done was to get rid of the function/class dualism in favor of a completely intensional definition with every piece of code as an intrinisc method. We had wanted that from the beginning, (and most of the code was already written that way). There were a variety of strong desires for a real intheriteance mechanism from Adele and me, from Larry Tesler, who was working on desktop publishing, and from the grad students. Dan had to find a better way than simula's ver rigid compile-time concpetion. It was time to make good on the idea that "everything was an object," which included all the internal "systems" objects like "activation records," etc. We were all agreed that the flexible syntax of the earlier Smalltalks was too flexible, and this level of extensibility was not desireable. All of the extensions we liked used various keyword schemes, so Dan came up with a combination keyword/operator syntax that was very flexible, but allowed the language to be read unambiguously by both humans and the machine. This allowed a FLEX machine-like byte-code compiler and efficient interpreter to be defined that ran up to 180 times as fast as the previous direct interpreter. The OOZE VM system could be modified to handle the new objects and its capactiy was well matched to

the ALTO's RAM and disk.

Inheritance

A word about inheritance. Simula-I had neither classes as objects nor inheritance. Simula-67 added the latter as a generalization to the ALGOL-60 <block> structure. This was a great idea. But it did have some drawbacks: minor ones like name clashes in multiple threaded lists (no on uses threaded lists anymore), and major ones like rigidity in the extended type structures, need to qualify types, only a single path of inheritance, and difficulty in adopting to an interactive development system with incremental cmpiling and other needs for instant changes. Then there were a host of problems that were really outside the scope of Simula's goals: having to do with various kinds of modeling and inferencing that were of interest in the world of artifical intelligence. For example, not all useful questions could be answered by following a static chain. Some of them required a kind of "inhertance" or "inferencing" through dynamcically bound "parts" (ie. instance variables). Multiple inerheritance also looked important but the corresponding possible clashes between methods of the same name in different superclases looked difficult to handle, and so forth.

On the other hand, since things can be done with a dynamic language that the difficult with a statically compiled one, I just decided to leave inhertance out as a feature in Smalltalk-72, knowing that we could simulate it back using Smalltalk's LISPlike flexibility. The biggest contributer to these AI ideas was Larry Tesler who used what is now called "slot inheritance" extensively in his various versions of early desktop publishing systems. Nowadays, this would be called a "delgation-style" inheritance scheme [Liberman 84]. Danny Bobrow and Terry Winograd during this period were designing a "frame-based" AI language called KRL which was "object-oriented" and I beleive was influenced by early Smalltalk. It had a kind of multiple inheritance--called *perspectives*--which permitted an object to play multiple roles in a very clean way. Many of these ideas a few years later went into PIE, an interesting extension of Smalltalk to networks andhigher level descriptions by Ira Goldstein and Bobrow [Goldstein & Bobrow 1980].

By the time Smalltalk-76 cam along, Dan Ingalis had come up with a scheme that was Simula-like in its semantics but could be incrementally changed on the fly to be in accord with our goals of close interaction. I was not completly thrilled with it because it seemed that we needed a better theory about ineritance entirely (and still do). For example, inheritance and instancing (which is a kind of inheritance) muddles both pragmatics (suc as factoring code to save space) and semantics (used for way too many tasks such as: specialization, generalization, speciation, etc.) Alan Borning employed a multiple inheritance scheme in Thinglab [Borning 1977] which was implemented in Smalltalk-76. But no comprehensive and clean multiple inerhitance scheme appeared that was compelling enough to surmount Dan's original Simula-like design.

Meanwhile, the running battle with Xerox continued. there were now about 500 ALTOs linked with Ethernets to each other and to Laserprinter and file servers, that used ALTOs as controllers. I wrote many memos to the Xerox planners trying to get them to make plans that included personal computing as one of their main directions. Here is an example:

A Simple Vision of the Future

A Brief Update Of My 1971 Pendery Paper

In the 1990's there will be millions of personal computers. They will be the size of notebooks of today, have high-resolution flat-screen reflective display.s, wigh less than ten pounds, have ten to twenty times the computing and storage capacity of an Alto. Let's call them Dynabooks.

The purchase price will be about that of a color television set of the era, although most of the machines will be given away by manufacturers who will be marketing the content rather than the container of personal computing.

...

Though the *Dynabook* will have considerable local storage and will do most computing locally, it will spend a large percentage of its time hooked to various large, global information utilities which will permit communication with others of ideas, data, working models, as well as the daily chit-chat that orgnizations need in order to function. The communications link will be by private and public wire and by packet radio, Dynabooks will also by used as servers in the information utilties. They will have enough power to be entirely shaped by solftware.

The Main Points Of This Vision

- There need only be a few hardware types to handle almost all of the processing activity of a system.
- Personal Computers, Communications Link, and Information Utilities are the three critical components of a Xerox future.

...

In other words, the *material* of a computer system is the computer irself, all of the *content* and *function* is fashioned in software.

There are two important guidelines to be drawn from this:

- Material: If the design and development of the hardware computer material is done as carefully and completely as Xerox's development of special light-sensitve alloys, then only one or two computer designs need to be built... Extra investment in devleopment here will be vastly repaid by simplifying the manufacturing process and providing lower costs through increased volume.
- Content: Aside from the wonderful generality of being able to continuously shape new content from the same material, *software* has three important characteristics:

- the *replication* time and cost of a content-function is zero
- the *development* time and cost for a content-function is high
- the *change* time and cost for a content-function is low

Xerox must take these several points seriously if it is to survive and prosper in its new business are of information media. If it does, the company has an excellent chance for several reasons:

- Xerox has the financial base to cover the large development costs of a small number of very powerful computer-types and a large number of software functions.
- Xerox has the marketing base tosell these functions on a wide enough scale to garner back to itself an incredible profit.
- Xerox has working for it an impressively large percentage of the best software designers in the world.

In 1976, Check Thacker designed the ALTO III that would use the new 16k chips and be able to fit on a desktop. It could be marketed for about what the large cumbersome special purpose "word-processors" cost, yet could do so much more. Nevertheless, in august of 1976, Xerox made a fateful decision: not to bring the ALTO III to market. This was a huge blow to many of us--even I, who had never really, really thought of the ALTO as anything but a stepping stone to the "real thing." In 1992, the world market for personal computers and workstations was $90 million--twice as much as the mainframe and mni market, and many times Xerox's 1992 gross. The most successful company of this era--Microsoft--is not a hardware company, but a software company.

The Smalltalk User Interface

I have been asked by several of the reviewers to say more about the development of the "Smalltalk-style" overlapping window user interface since thee are now more than 20 million computers in the workld that use its descendents. A decent history would be as long as this chapter, and none has been written so far. There is a summary of some of the ideas in [Kay 89]--let me add a few more points.

All of the elements eventually use in the Smalltalk user interface were already to be found in the sixties--as different ways to access and invoke the functionality provided by an interactive system. The two major centers of ieads were Lincoln Labs and RAND corp--both ARPA funded. The big shift that consolidated these ideas into a powerful theory and long-lived examples came because the LRG focus was on children. Hence,we were thinking about learning as being one of the main effects we wanted to have happen. EArly on, this led to a 90 degree rotation of the purposed of the user interface from"access to functionality" to "environment in which users learn by doing." This new stance could now respond to the echos of Montessori and Dewey, particularly the former, and got me, on rereading Jerome Bruner, to think beyond the children's curriculum to a "curriculum of the user interface."

The particular aim of LRG was to find the equivalent of writing--that is learning and

thinking by doing in a medium--our new "pocket universe." For various reasons I had settled on "iconic programming" as the way to achieve this, drawing on the iconic representations used by many ARPA projects in the sixties. My friend Nicholas Negroponte, an architect, was extremely interested in how environments affected peoples' work and creativity. He was interested in embedding the new computer majic in familiar surroundings. I had quite a bit of theatrical experience in a past life, and remembered Coleridge's adage that "people attend 'bad theatre' hoping to forget, people attend 'good theatre' *aching to remember*." In other words, it is the ability to evoke the audience's own intelligence and experiences that makes theatre work.

Putting all this together, we want an apparently free environment in which exploration causes desired sequences to happen (Montessori); one that allows kinesthetic, iconic, and symbolic learning--"*doing* with *images* makes *symbols*" (Piaget & Bruner); the user is never trapped in a mode (GRAIL); tha magic is embedded in the familiar (negroponte); and which acts as a magnifying mirror for the user's own intelligence (Coleridge). It would be a great finish to ths story to say that having articulated this we were able to move straightforwardly to the design as we know it today. In fact, the UI design work happened in fits and starts in between feeding Smalltalk itself, designing children's experiments, trying to understand iconic construction, and ust playing around. In spite of this meandering, the context almost forced a good design to turn out anyway. Just about everyone at PARC at this time had opinions about the UI, ours and theirs. It is impossible to give detailed credit for the hundreds of ideas and discussions. However, the consolidation can certainly be attributed to Dan Ingalls, for listening to everyone, contributing original ideas, and constantly building a design for user testing. I had a fiar amount to do with setting the context, invetning overlapping windows, etc., and Adele and I designed most of the experiments. Beyond that, Ted Kaeh;er, and vistor Ron Baecker made highly valuable contrubutions. Dave Smith designed, SmallStar, the protype iconic interface for the Xerox Star product [Smith 83].

Meanwhile, I had gotton Doug Fairbairn interested in the *Notetaker*. He designed a wonderful "smart bus" that could efficiently handle slow multiple processors and the system looked very promising, even though most of the rest of PARC thought I was nuts to abandon the fast bipolar hw of the ALTO. But I couldn't see that bipolar was ever going to make it into a laptop or Dynabook. On the other hand I hated the 8-bit micros that were just starting to appear, because of the silliness and naivete of their designs--there was no hint that anyone who had ever designed software was involved.

Smalltalk-76

Dan finished the Smalltalk-76 design November, and he, Dave Robson, Ted Kaehler, and Diana Merry, successfully implemented the system from scratch (which included rewriting all of the existing class definition) in just seven months. this was such a wonderful achievement that I was bowled over in spite of my wanting to start over. It was fast, lively, could handle "big" problems, and was great fun. The system consisted of about 50 classes descibed in about 180 pages of source code. This included all of the OS functions, files, printing and other Ethernet services, the window interface, editors, graphics and painting systems, and two new contributions by Larry Tesler, the famous browsers for static methods in the inheritance heirarchy and dynamic contexts for debuging in the runtime environment. In every way it was the consolidation of all of our ideas and yearning about Smalltalk in one integrated package. Al Smalltalks since have resmbled this conception very closely. In many ways, as Tony Hoare once remakred about Algol, Dan's Smalltalk-76 was a great improvement on its successors!

Here are two stylish ST-76 classes written by Dan.

```
Class new title: 'Window';
  fields: 'frame';
  asFollows!
This is a superclass for
presenting windows on the
display. It holds control until
the stylus is depressed outside.
While it holds control, it
distributes messages to itself
based on user actions.
Scheduling
startup
  [frame contains; stylus =>
    self enter.
    repeat:
      [frame contains: stylus =>
        [keyboard active => [ self
keyboard ]
        stylus down => [ self
pendown ]]
      self outside => []
      stylus down => [ ^self
leave ]]]
  ^false]
Default Event Responses
enter [self show]
leave
outside [ ^false]
pendown
keyboard [ keyboard next.
frame flash ]
Image
show
  [frame outline: 2.
titleframe put: self title at: frame
origina + title loc.
titleframe complement]
... etc.
```

```
Class new title: 'DocWindow';
  subclassOf: Window;
  fields: 'document scrollbar
editMenu';
  asFollows!
User events are passed on to the
document while the window is
active. If the stylus goes out of
the window, scrollbar and the
editMenu are each given a
chance to gain control. Event
Responses
enter [ self show. edit Menu
show. scrollbar show ]
leave [ document hideselection.
editMenu hide. scrollbar hide ]
outside
  [editMenu startup => []
  scrollbar startup => [self
showdoc]
  ^false]
pendown [ document pendown ]
keyboard [ document
keyboard ]
Image
show [ super show. self
showDoc ]
showDoc [ document showin;
frame at: scrollbar position ]
title [^document title]
```

Notice, particularly in class Window, how the code is expressed as goals for other objects (or itself) to achieve. The superclass Windpow's main job is to notice events and distribute them as messages to its subclasses. In the example, a document window (a subclass of DocWindow) is going to deal with the effects of user interactions. The Window class will notice that the keyboard is actie and send a message to itself which will be intercepted by the subclass method. If there is no method the character will be thrown away and the window will flash. In this case, it find DocWindow method: keyboard, which tells the held document to check it out.

In January of 1978 Smalltalk-76 had its first real test. CSL had invited the top ten executives of Xerox to PARC for a two day seminar on software, with a special emphasis on complexity and what could be done about it. LRG got asked to give them a hands-on experience in end-user programming so "they could do 'something real' over two 1 1/2 hour sessions." We immediately decided not to teach them Smalltalk-76 (my "burn our disk packs" point in spades), but to create in two months in Smalltalk-76 a rich system especially tailored for adult nonexpert users (Dan's point in trumps). We took our "Simpula" job shop simulation model as a starting point and decided to build a user interface for a generalized job shop simulation tool that the executives could make into specific dynamic simulations that would act out their changing states by animating graphics on the screen. We called it the Smalltalk SimKit. This was a maximum effort and everyone pitched in. Adele became the design leader in spite of the very recent appearence of a new baby. I have a priceless memory of her debugging away on the SimKit while simultaneously nursing Rachell.

There were many interesting problems to be solved. The system itself was straightforward but it had to be completely sealed off from Smalltalk proper, particularly with regard to error messages. Dave Robson came up with a nice scheme (almost an expert system) to capture complaints from the bowels of Smalltalk and translated them into meaningful SimKit terms. There were many user interface details--some workaday, like making new browsers that could only look at the four SimKit classes (Station, Worker, Job, Report), and some more surprising as when we tried it on ten PARC nontechnical adults of about the same age and found that they couldn't read the screen very well. The small fonts our thirtysomething year-old eyes were used to didn't work for those in their 50s. This led to a nice introduction to the system in which the executives were encouraged to customize the screen by choosing amoung different fonts and sizes with the side effect that they learned how to use the mouse unselfconsciously.

On the morning of the "big day" Ted Kaehler decided to make a change in the virtual memory system OOZE to speed it up a little. We all held our breaths, but such was the clarity of the design and the confidence of the implementers that it did work, and the executive hands-on was a howling success. About an hour into the first session one of the VPs (who had written a few programs in FORTRAN 15 years before) finally realized he was programming and mused "so it's finally come to this." Nine out of the ten executives were able to finish a simulation problem that related to their specific interests. One of the most interesting and sophisticated was a PC board production line done by the head of a Xerox owned company using actual figures (that he carried around in his head) to prime a model that could not be solved easily by closed form mathematics--it revealed a seriousl flaw in the dispoition of workers given the line's average probability of manufacturing defects.

Another important system done at this time was Alan Borning's Thinglab [Borning 1979]--the first serious attempt to go beyond Ivan Sutherland's Sketchpad. Alan devised a very nice approach for dealing with constraints that did not require the solver to be omnicient (or able to solve Fermat's last theorem).

Meanwhile, the *NoteTaker* was getting realler, bigger, and slower. By this time the Western Digital emulation-style chips I hoped to used showed signs of being "diffusion-ware," and did not look like they would really show up. We started looking around for something that we could count on, even if it didn't have a good architecture. In 1978, the best candidate was the Intel 8086, a 16-bit chip (with many unfortunate remnants of the 8008 and 8080), but with (barely) enough capacity to do the job--we would need three of them to make up

for the ALTO, one for the interpreter, one for bitmapped graphics, and one for i/o (networking, etc).

Dan had been interested in the *NoteTaker* all along and wanted to see if the could make a version of Smalltalk-76 that could be the *NoteTaker* system. IN order for this to happen it would have to run in 256K (the maximum amount of RAM that we had planned for the machine. None of the NOVA-like emulated "machine-code" from the ALTO could be brought over, and it had to fit in memory as well-lthere would only be floppies, no swapping memory existed. This challenge led to some excellent improvements in the system design. Ted Kaehler's system tracer (which could write out new virual memories from old ones) was used to clone Smalltalk-76 into the *NoteTaker*. The indexed object table (as was used in early Smalltalk-80) first appeared here to simplify object access. An experiment in stacking contexts contiguously was tried: to save space and gain speed. Most of the old machine code was rewritten in Smalltalk and the total machine kernal was reduced to 6K bytes of (the not very strong) 8086 code.

All of the re-engineering had an interesting effect. Through the 8086 was not as good at bitblt as the ALTO (and much of the former machine code to assist graphics was now in Smalltalk), the overall interpreter was about twice as fast as the ALTO version (because not all the Smalltalk byte-code interpreer would fit into the 4k microcode memory on the ALTO). With various kinds of tricks and tuning, graphics display was "largely compensated" (in Dan's words). This was mainly because the ALTO did not have enough microcode memory to take in all of the Smalltalk emulation code--some of it had to be rendered in emulated "NOVA" code which forced two layers of interpretation. In fact, the *Notetaker* worked extreemely well, though it would have crushed any lap. It had hopped back on the desk, and looked suspiciously like miniCOM (and several computers that would appear a few years later). It really did run on batteries and several of us had the pleasure of taking *NoteTaker* on a plane and running an object-oriented system with a windowed interface at 35,000 feet.

We eventually built about 10 of the machines, and though in many senses an engineering success, what had to be done to make them had once again squeezed out the real end-users for whom it was originally aimed. If Xerox (and PARC) as a whole had believved in these smaller scale ideas, we could have put much more silicon muscle behind the dreams and successfully built them in the 70's when they were first possible. It was a bitter disappointment to have to get the wrong kind of CPU from Intel and the wrong kind of display from HP because there was not enough corporate will to take advantage of internal technological expertise.

By now it was already 1979, and we found ourselves doing one of our many demos, but this time for a very interested audience: Steve Jobs, JeffRaskin, and other technical people from Apple. They had started a project called *Lisa* but weren't quite sure what it shouldbe like, until Jeff said to Steve, "You should really come over to PARC and see what they ae doing." Thus, more than eight years after overlapping windows had been invented and more than six years after the ALTO started running, the people who could really do something about the ideas, finally to to see them. The machine used was the Dorado, a very fast "big brother" of the ALTO, whose Smalltalk microcode had been largely written by Bruce Horn, one of our original "Smalltalk kids" who was still only a teen-ager. Larry Tesler gave the main part of the demo with Dan sitting in the copilot's chair and Adele and I watched from the rear. One of the best parts of the demo was when Steve Jobs said he didn't like the blt-style scrolling we were using and asked if we cold do it in a smooth continuous style. In less

than a minute Dan found the methods involved, made the (relatively major) changes and scrolling was now continuous! This shocked the visitors, espeicially the programmers among them, as they had never seen a really powerful incremental system before.

Steve tried to get and/or buy the technology from Xerox (which was one of Apple's minority venture captialists), but Xerox would neither part with it nor would come up with the resources to continue to develop it in house by funding a better *NoteTaker* cum Smalltalk.

VI. 1980-83--The release version of Smalltalk (-80)

The greatest sin in Art is not Boredom,
as is commonly supposed, but lack of
Proportion"
-- Paul Hindemith

As Dan said "the decision not to continue the *NoteTaker* project added motivation to release Smalltalk widely." But not for me. By this time I was both happy about the cleanliness and elegance of the Smalltalk conception as realized by Dan and theothers, and sad that it was farther away than ever from Children--it came to me as a shock that no child had programmed in any Smalltalk since Smalltalk-76 made its debut. Xerox (and PARC) were now into "workstations" as things in themselves--but I still wanted "playstations". The romance of the Dynabook seemed less within grasp, paradoxically just when the various needed technologies were starting to be commercially feasible--some of them, unfortunately, like the flat-screen display, abandoned to the Japanese by the US companies who had invented them. This was a major case of "snatching defeat from the jaws of victory." Larry Tesler decided that Xerox was never going to "get it" and was hired by Steve Jobs in May 1980 to be principal designer of the *Lisa* I agreed, had a sabbatical coming, and took it.

Adele decided to drive the documentation and release process for a new Smalltalk that could be distributed widely almost regardless of the target hardware. ONly a few changes had to be made to the NoteTaker Smalltalk-78 to make a releasable sysstem. Perhaps the change that was most ironic was to turn the custom fonts that made Smalltalk more readable (and were a hallmark of the entire PARC culture) back into standard pedestrian ASCII characters. According to Peter Deutsch this "met with heated opposition within the group at the time, but has turned out to be essential for the acceptance of the system in the world." Another change was to make blocks more like lambda expressions which, as Peter Deutsch was to observe nine yeas later: "In retrospect, this proliferation of different kinds of instantiations and scoping was probably a bad idea." The most puzzling strange idea--at least to me as a new outsider--was the introduction of metaclasses (really just to make instance initialization a little easier--a very minor improvement over what Smalltalk-76 did quite reasonably already). Peter's 1989 comment is typical and true: "metaclasses have proven confusing to many users, and perhaps in the balance more confusing than valuable." In fact, in their PIE system, Goldstein and Bobrow had already implemented in Smalltalk on "observer language", somewhat following the view-oriented approach Ihad been advocating and in some ways like the "perspectives" proposed in KRL [Goldstein *]. Once one can view an instance via multiple perspectives even "sem-metaclasses" like Class Class and Class Object are not really necessary since the object-role and instance-of-a-class-role are just different views and it is easy to deal with life-history issues includeding

instantiation. This was there for the taking (along with quite a few other good ideas), but it wsn't adopted. My guess is that Smalltalk had moved into the final phase I memntioned at the beginning of this story, in which a way of doing things finally gets canonized into an inflexible belief structure.

Coda

One final comment. Hardware is really just software crystallized early. It is there to make program schemes run as efficiently as possible. But far too often the hardware has been presented as a given and it is up to software designers to make it appear reasonable. This has caused low-level techniques and excessive optimization to hold back progress in program design. As Bob Barton used to say: "Systems programmers are high priests of a low cult."

One way to think about progress in software is that a lot of it has been about finding ways to *late-bind*, then waging campaigns to convince manufacturers to build the ideas into hardware. Early hardware had wired programs and parameters; random access memory was a scheme to late-bind them. Looping and indexing used to be done by address modification in storiage; index registers were a way to late-bind. Over the years software designers have found ways to late-bind the locations of computations--this led to base/bounds registers, segment relocation, page MMUs, migratory processes, and so forth. Time-sharing was held back for years because it was "inefficient"-- but the manufacturers wouldn't put MMUs on the machines, universities had to do it themselves! Recursion late-binds parameters to procedures, but it took years to get even rudimentary stack mechanisms into CPUs. Most machines still have no support for dynamic allocation and garbage collection and so forth. In short, most hardware designs today are just re-optimizations of moribund architectures.

From the late-binding perspective, OOP an be viewed as a comprehensive technique for late-binding as many things as possible: the *mix* of state and process in a set of behaviors, *where* they are located, *what* they are called, *when* and *why* the are invoked, *which* JW is used, etc., and more subtle, the *strategies* used in the OOP scheme itself. The art of the wrap is the art of the trap.

Consider the two ases that must be handled efficiently in order to completely wrap objects. It would be terrible if *a* + *b* incurred any overhead if *a* and *b* were bound, say, to "3" and "4" ina form that could be handled by the ALU. The operations should occur full speed using look-aside logic (in the simplest scheme a single *and* gate) to trap if the operands aren't compatible with the ALU. Now all elementary operations that have to happen fast have been wrapped without slowind down the machine.

The second case happens if the trap has determined the objects in questions are too complicated for the ALU. Now the HW has to dynamically find a method that can handled the objects. This is very simiilar to indexing--the class of one of the objects is "indexed" by the desired method-selector in a slightly more general way. In other words the *virtual-address* of a method is <class><selector>. Since most HW today does a virtual address translation of some kind to find the real address--a trap--it is quite possible to hide the overhead of the OOP dispatch in the MMU overhead that has already been rationalized.

Again, the whole point of OOP is not to have to worry about what is *inside* an object. Objects made on different machines and with different languages *should* be able to talk to

each other--and will have-to in the future. Late-binding here involves trapping incmpatibilities into recompatibility methods--a good discussion of some of the issues is found in [Popek 1984].

Staying with the metaphor of late-binding, what further late-binding schemes might we expect to see? One of the nicest late-binding schemes that is being experimented with is the *metaobject protocol* work at Xerox PARC [Kiczales 1991]. The notion is that the language designer's choice for the internal representation of instances, variables, etc., may not cover wht the implementer needs, so within a fixed semantics they allow the implmenter to give the system strategies--for example, using a hashed lookup for slots in an instance instead of direct indexing. These are then efficiently compiled and extend the base implemenation of the sustem. This is a direct descendant of similar directions from the past of Simula, FLEX, CDL, Smalltalk, and Actors.

Another late-binding scheme that is already necessary is to get away from directo protocol matching when a new object shows up in a system of objects. In other words, if someone sends you an object from halfway around the world it will be unusual if it conforms to your local protocols. At some point it will be easier to have it carry even more information about itself--enough so its speicifications can be "understood" and its confiugration into your mix done by the more subtle mathcing of *inference*.

A look beyond OOP as we know it today can also be done by thinking about late-binding. Prolog's great idea is that it doesn't need binding to values in order to carry out computations [Col **]. The variable is an object and a web of partial results can be built to be filled in when a binding is finally found. Eurisko [Lenat **] constructs its methods--and modifies its basic strategies--as it tries to solve a problem. Instead of a problem looking for methods, the methods look for problems--and Eurisko looks for the methods of the methods. This has been called "opportunistic programming"--I think of it as a drive for more enlightenment, in which problems get resolved as part of the process.

This higher computational finesse will be needed as the next paradigm shift--that of pervasive networking--takes place over the next five years. Objects will gradually become active agents and will travel the networks in search of useful information and tools for their managers. Objects brought back into a computational environment from halfway around the world will not be able to configure themselves by direct protocol matching as do objects today. Intead, the objects will carry much more informration about themselves in a form that permits *inferential* docking. Some of the ongoing work in specificaion can be turned to this task [Guttag **][Goguen **].

Toungue in cheek, I once characterized progress in programming languages as kind of "sunspot" theory, in which major advances took place about every 11 years. We started with machine code in 1950, then in 1956 FORTRAN came along as a "better old thing" which if looked at as "almost a new thing" became the precursor of ALGOL-60 in 1961. IN 1966, SIMULA was the "better old thing," which if looked at as "almost a new thing" became the precursor to Smalltalk in 1972.

Everything seemed set up to confirm the "theory" once more: in 1978 Eurisko was in place as the "better old thing" that was "almost a new thing". But 1983--and the whole decade--came and went without the "new thing". Of course, such a theory is silly anyway--and yet, I think the enormous commercialization of personal computering has smothered much of the kind of work that used to go on in universities and research labs, by sucking the talented

kids towards practical applications. With companies so risk-adverse towards doing their own HW, and the HW companies betraying no real understanding of SW, the result has been a great step backwards in most respects.

A twentieth century problem is that technology has become too "easy". When it was hard to do *anything* whether good or bad, enough time was taken so that the result was usually good. Now we can make things almost trivially, especially in software, but most of the designs are trivial as well. This is inverse vandalism: the making of things because you can. Couple this to even less sophisticated buyers and you have generated an exploitation marketplace similar to that set up for teenagers. A counter to this is to generate enormous disatisfaction with one's designs using the entire history of human art as a standard and goal. Then the trick is to decouple the disatisfaction from self worth--otherwise it is either too depressing or one stops too soon with trivial results.

I will leave the story of early Smalltalk in 1981 when an extensive series of articles on Smalltalk-80 was published in *Byte* magazine, [Byte 1981] followed by Adele's and Dave Robsons books [Goldberg 1983] and the official release of the system in 1983. Now programmers could easily implement the virtual machine without having to reinvent it, and, in several cases, groups were able to roll their own *image* of basic classes. In spite of having to run almost everywhere on moribund HW architectures, Smalltalk has proliferated amazingly well (in part because of tremendous optimization efforts on these machines) [Deutsch 83]. As far as I can tell, it still seems to be the most wiidely used system that claims to be object-oriented. It is incredible to me that no one since has come up with a qualitatively better idea that is as simple, elegant, easy to program, practical, and comprehensive. (It's a pity that we didn't know about PROLOG then or vice versa, the combinations of the two languages done subsequently are quite intriguing).

While justly applauding Dan, Adele and the others that made Smalltalk possible, we must wonder at the same time: where are the Dans and the Adeles of the '80s and '90s that will take us to the next stage?

Access DB# _____

SEARCH REQUEST FORM

Scientific and Technical Information Center

Requester's Full Name: Todd Ingberg Examiner # : 75084 Date: 7/14/05
Art Unit: 2193 Phone Number 23723 Serial Number: 09/184,738
Mail Box and Bldg/Room Location: RAN 5B09 Results Format Preferred (circle): PAPER DISK E-MAIL

If more than one search is submitted, please prioritize searches in order of need.

****

Please provide a detailed statement of the search topic, and describe as specifically as possible the subject matter to be searched. Include the elected species or structures, keywords, synonyms, acronyms, and registry numbers, and combine with the concept or utility of the invention. Define any terms that may have a special meaning. Give examples or relevant citations, authors, etc, if known. Please attach a copy of the cover sheet, pertinent claims, and abstract.

Title of Invention: _____

Inventors (please provide full names): _____

_____

Earliest Priority Filing Date: NOV 15, 1994

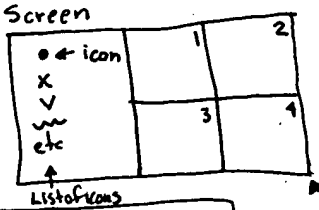
**For Sequence Searches Only* Please include all pertinent information (parent, child, divisional, or issued patent numbers) along with the appropriate serial number.*

AREAS of Technology → DRAG AND DROP
→ Iconic Programming

Tool as in well known practice in the Art

Wrapper

Basic anatomy of Art



part or all icons make up a dragable list

areas of development environment make up drag to area list

Each area could have its own dragable list

when the user selects an icon the selected object is con wrapped. See the over all concept of what a wrapper is from Taylor. Example is about migration of systems not dragging and dropping. Looking for technology behind iconic programming.

****

STAFF USE ONLY	**Type of Search**	**Vendors and cost where applicable**
Searcher: _____	NA Sequence (#)_____	STN _____
Searcher Phone #: _____	AA Sequence (#)_____	Dialog _____

Freeform Search

Database: US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Term: L4 AND wrapper

Display: 50 **Documents in Display Format:** REV **Starting with Number** 1

Generate: ○ **Hit List** ◉ **Hit Count** ○ **Side by Side** ○ **Image**

Search Clear Interrupt

Search History

DATE: Thursday, July 14, 2005 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=NO; OP=OR			
L5	L4 AND wrapper	30	L5
L4	L3 AND (drag NEAR drop)	638	L4
L3	L2 AND drag	1395	L3
L2	L1 AND (visual OR iconic) AND (object)	7111	L2
L1	703/$$$.ccls. OR 707/$$$.ccls. OR 715/$$$.ccls. OR 717/$$$.ccls.	34748	L1

END OF SEARCH HISTORY

Access DB# 159340

SEARCH REQUEST FORM

Scientific and Technical Information Center

52

Requester's Full Name: Todd Ingberg Examiner #: 75084 Date: 7/14/05
Art Unit: 2193 Phone Number 23723 Serial Number: 09/184,738
Mail Box and Bldg/Room Location: RAN 5B09 Results Format Preferred (circle): PAPER DISK E-MAIL

If more than one search is submitted, please prioritize searches in order of need.

****

Please provide a detailed statement of the search topic, and describe as specifically as possible the subject matter to be searched. Include the elected species or structures, keywords, synonyms, acronyms, and registry numbers, and combine with the concept or utility of the invention. Define any terms that may have a special meaning. Give examples or relevant citations, authors, etc, if known. Please attach a copy of the cover sheet, pertinent claims, and abstract.

Title of Invention: _____

Inventors (please provide full names): _____

_____

Earliest Priority Filing Date: NOV 15, 1994

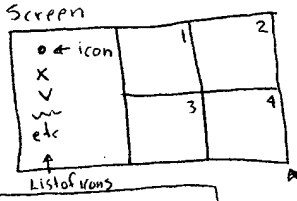
***For Sequence Searches Only* Please include all pertinent information (parent, child, divisional, or issued patent numbers) along with the appropriate serial number.**

AREAS of Technology → DRAG AND DROP
→ Iconic Programming

Tool as in well known practice in the Art

Wrapper

Basic anatomy of Art



areas of development environment make up drag to area list

part or all icons make up a dragable list

Each area could have its own dragable list

when the user selects an icon the selected object is ~~con~~ wrapped. See the over all concept of what a wrapper is from Taylor. Example is about migration of systems not dragging and dropping. Looking for technology behind iconic programming.

****

STAFF USE ONLY	Type of Search	Vendors and cost where applicable
Searcher: Holloway	NA Sequence (#)	STN
Searcher Phone #: 2-3528	AA Sequence (#)	Dialog ✓
Searcher Location: RND 4B19	Structure (#)	Questel/Orbit
Date Searcher Picked Up: 7-20-05	Bibliographic ✓	Dr.Link
Date Completed: 7-20-05	Litigation	Lexis/Nexis
Searcher Prep & Review Time: 60	Fulltext ✓	Sequence Systems
Clerical Prep Time:	Patent Family	WWW/Internet ✓
Online Time: 20	Other	Other (specify)

RECEIVED
JUL 14 2005
BY:

PTO-1590 (8-01)

STIC Search Report
EIC 2100

STIC Database Tracking Number: 153340

TO:	From:
Todd Ingberg **Location: RND 5B09** **Art Unit : 2193** **Wednesday, July 20, 2005** **Case Serial Number: 09/184738**	**David Holloway** **Location: EIC 2100** **RND 4B19** **Phone: 2-3528** **david.holloway@uspto.gov**

Search Notes

Dear Examiner Ingberg,

Attached please find your search results for above-referenced case.
Please contact me if you have any questions or would like a re-focused search.

David

30/5/20 (Item 20 from file: 350)
DIALOG(R)File 350:Derwent WPIX

008894800
WPI Acc No: 1992-022069/199203
XRPX Acc No: N92-016734
Display item selection visual indicator - allows ICON selection **in which shrink** wrap **grey and black dithered outline appears around opaque pels of ICON**
Patent Assignee: ANONYMOUS (ANON)
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week
RD 332055	A	19911210				199203 B

Priority Applications (No Type Date): RD 91332055 A 19911120

Abstract (Basic): RD 332055 A
The **selection** visual **indicator** allows **icon selection** and **icon** label editing. For **selection** , a `shrink **wrap** ' grey and black dithered outline appears around the opaque pels of the **icon** . For `in-use' **indications** , a background square of dithered light grey and yellow appears as the in-use visual.
A dashed or dotted line is incorporated around the **object** . To **select** the **icon** label for editing, the user invokes text edit for the entry area with a cursor indicating the position to type text. The `target' visual appears when an object is being dragged to another object. It is a black box around the object. It appears when the mouse button is held down, or when the drag task is still being performed.
ADVANTAGE - Conveys message neatly and cleanly without demanding too much attention, as would swatch of colour and adding capability to show `in-use' `cursor-selected' and `target' increases amount of function. (1pp Dwg.No.0/0
Title Terms: DISPLAY; ITEM; SELECT; VISUAL; INDICATE; ALLOW; SELECT; SHRINK ; **WRAP** ; GREY; BLACK; DITHER; OUTLINE; APPEAR; OPAQUE
Derwent Class: T01
International Patent Class (Additional): **G06F-000/01**
File Segment: EPI

```
Set     Items   Description
S1        954   AU=(MORRIS R? OR MORRIS, R?)
S2         24   AU=(DENTON L? OR DENTON, L?)
S3          5   S1 AND S2
S4        137   (S1 OR S2) AND (ICONIC OR ICON OR ICONS OR WRAPPER? OR MUL-
              TIPLE()VIEW? ? OR OBJECT? OR OOP? ? OR ENVELOPE? OR DRAG(N)(D-
              ROPS OR DROPPED OR DROPPING OR DROP) OR GUI OR GUIS)
S5          1   S4 AND IC=G06F-007
S6         23   S4 AND IC=(G06F OR H04L)
S7          1   S4 AND (TIMELINE? OR MAP OR MAPPED OR SCRIPT? OR OUTPUT)(N-
              )VIEW?
S8         26   S3 OR S5 OR S6 OR S7
S9         26   IDPAT (sorted in duplicate/non-duplicate order)
S10        20   IDPAT (primary/non-duplicate records only)
File 344:Chinese Patents Abs Aug 1985-2005/May
         (c) 2005 European Patent Office
File 347:JAPIO Nov 1976-2005/Feb(Updated 050606)
         (c) 2005 JPO & JAPIO
File 348:EUROPEAN PATENTS 1978-2005/Jul W02
         (c) 2005 European Patent Office
File 349:PCT FULLTEXT 1979-2005/UB=20050714,UT=20050707
         (c) 2005 WIPO/Univentio
File 350:Derwent WPIX 1963-2005/UD,UM &UP=200545
         (c) 2005  Thomson Derwent
```

10/5/4 (Item 4 from file: 350)
DIALOG(R)File 350:Derwent WPIX

012324642 **Image available**
WPI Acc No: 1999-130748/199911
XRPX Acc No: N99-095179
Computer implemented application development system - sets properties of additional objects **until all of desired** objects **have been specified, and then interconnecting** objects **in temporal sequences**
Patent Assignee: DENTON L E (DENT-I); MORRIS R M (MORR-I)
Inventor: **DENTON L E** ; **MORRIS R M**
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
US 5862372	A	19990119	US 94340702	A	19941116	199911	B

Priority Applications (No Type Date): US 94340702 A 19941116
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
US 5862372	A		11	G06F-009/06	

Abstract (Basic): US 5862372 A
NOVELTY - The **objects** written at standard specification are wrapped. Four views are established and synchronized. The **object** is moved into one of the four views, and properties are set to the **object** . The properties are set to additional **objects** until all the desired **objects** have been specified. The **objects** are interconnected in temporal sequences. The flow of data and control between **objects** is

specified. A script reflector is generated and the script is executed.
USE - For authoring application system.
ADVANTAGE - **Icons** representing the **objects** are placed into appropriate view, even though user does not know how to write specialized code. DESCRIPTION OF DRAWING(S) - The figure shows simple program for four **views** , **output** , map, multitrack and work form simultaneously displayed in separate windows.
Dwg.5/6
Title Terms: COMPUTER; IMPLEMENT; APPLY; DEVELOP; SYSTEM; SET; PROPERTIES; ADD; **OBJECT** ; **OBJECT** ; SPECIFIED; INTERCONNECT; **OBJECT** ; TEMPORAL; SEQUENCE
Derwent Class: T01
International Patent Class (Main): **G06F-009/06**
International Patent Class (Additional): **G06F-009/22**
File Segment: EPI

19/3,K/26 (Item 4 from file: 47)
DIALOG(R)File 47:Gale Group Magazine DB(TM)

02581045 SUPPLIER NUMBER: 03360150 (USE FORMAT 7 OR 9 FOR FULL TEXT)
Jack 2; sprints to the fore. (evaluation)
Baras, Edward
PC Magazine, v3, p139(6)
July 24, 1984
DOCUMENT TYPE: evaluation LANGUAGE: ENGLISH RECORD TYPE: FULLTEXT
WORD COUNT: 2958 LINE COUNT: 00222

... then pressing the Enter key. Alternatively, you can type the first letter of the command.

Selecting ZOOM brings you to another screen of **icons** , the Envelope **screen** (see figure 3). Like the Disk **screen** , the **Envelope screen** consists of a command line, a hint line, and a lineup of **icons** . The **icons** in this case are **envelopes** , Jack2's equivalent of files. Any function or combination of functions, whether word processing, database, spreadsheet, or graphics, can be contained in one **envelope** . A disk **icon** on the right side of the screen shows which disk has been opened for inspection...you used as a template for your data entry. There is also a blank record **icon** , **indicated** by an arrow. To fill out a form of data, you simply point the arrow...
...some other field.

The integration between database and word processing is nicely orchestrated. You can **create** one **envelope** with a form containing a document, such as a form letter to be sent to...

```
Set     Items   Description
S1    3289057   OBJECT OR OBJECTS OR ICON OR ICONOGRAPHIC OR ICONS OR SYMB-
             OL OR SYMBOLS OR THUMBNAIL? OR (GUI OR GUIS OR GRAPHICAL()USE-
             R()INTERFACE?)(2N)(FILE? OR ONSCREEN()REPRESENTATION? OR PICT-
             URE? OR IMAGE?)
S2   18086256   CLICK? OR SELECT? OR DRAG? ? OR INDICAT? OR CHOOSE? OR PIC-
             K? OR INSTANTIAT? OR ACTIVAT?
S3    1555784   WRAPPER OR WRAPPERS OR WRAPS OR WRAP OR WRAPPING OR ENVELO-
             P? OR CONTAINER?
S4       1913   S1 (10N) S2 (10N) S3
S5        378   S1(3N)S2 AND S3(2N)(CREATE? OR CREATING OR GENERAT? OR AUT-
             HORING OR AUTHORS OR WRITES OR LAUNCHES OR SPAWNS)
S6     103138   S1(2N)S2
S7         91   S4 AND S5
S8        419   S6 (10N) S3
S9        359   S4 (8N) (ONSCREEN? OR GUI OR GRAPHIC? OR DISPLAY? ? OR SCR-
             EEN? ? OR DESKTOP?)
S10      1017   S5 OR S7 OR S8 OR S9
S11       580   RD (unique items)
S12       233   S11 NOT PY>1994
S13        19   S8 AND S9 AND S12
S14       394   S5 OR S7 OR S13
S15      5243   S1(2N)(S3 OR WRAPPED)
S16        93   S15 AND S12
S17        18   S7 AND S16
S18        35   S13 OR S17
S19        35   RD (unique items)
File 275:Gale Group Computer DB(TM) 1983-2005/Jul 20
         (c) 2005 The Gale Group
File  47:Gale Group Magazine DB(TM) 1959-2005/Jul 20
         (c) 2005 The Gale group
File  75:TGG Management Contents(R) 86-2005/Jul W2
         (c) 2005 The Gale Group
File 636:Gale Group Newsletter DB(TM) 1987-2005/Jul 19
         (c) 2005 The Gale Group
File  16:Gale Group PROMT(R) 1990-2005/Jul 19
         (c) 2005 The Gale Group
File 624:McGraw-Hill Publications 1985-2005/Jul 20
         (c) 2005 McGraw-Hill Co. Inc
File 484:Periodical Abs Plustext 1986-2005/Jul W3
         (c) 2005 ProQuest
File 813:PR Newswire 1987-1999/Apr 30
         (c) 1999 PR Newswire Association Inc
File 141:Readers Guide 1983-2004/Dec
         (c) 2005 The HW Wilson Co
File 696:DIALOG Telecom. Newsletters 1995-2005/Jun 20
         (c) 2005 The Dialog Corp.
File 553:Wilson Bus. Abs. FullText 1982-2004/Dec
         (c) 2005 The HW Wilson Co
File 621:Gale Group New Prod.Annou.(R) 1985-2005/Jul 20
         (c) 2005 The Gale Group
File 674:Computer News Fulltext 1989-2005/Jul W3
         (c) 2005 IDG Communications
File  88:Gale Group Business A.R.T.S. 1976-2005/Jul 19
         (c) 2005 The Gale Group
File 160:Gale Group PROMT(R) 1972-1989
         (c) 1999 The Gale Group
File 635:Business Dateline(R) 1985-2005/Jul 20
         (c) 2005 ProQuest Info&Learning
File  15:ABI/Inform(R) 1971-2005/Jul 20
         (c) 2005 ProQuest Info&Learning
File  13:BAMP 2005/Jul W2
         (c) 2005  The Gale Group
File 810:Business Wire 1986-1999/Feb 28
         (c) 1999 Business Wire
```

File 647:CMP Computer Fulltext 1988-2005/Jul W1
(c) 2005 CMP Media, LLC
File 98:General Sci Abs/Full-Text 1984-2004/Dec
(c) 2005 The HW Wilson Co.
File 148:Gale Group Trade & Industry DB 1976-2005/Jul 20
(c)2005 The Gale Group
File 634:San Jose Mercury Jun 1985-2005/Jul 19
(c) 2005 San Jose Mercury News
File 20:Dialog Global Reporter 1997-2005/Jul 20
(c) 2005 The Dialog Corp.

19/3,K/1 (Item 1 from file: 275)
DIALOG(R)File 275:Gale Group Computer DB(TM)

01711288 SUPPLIER NUMBER: 16197466 (USE FORMAT 7 OR 9 FOR FULL TEXT)
Building component software with visual C++ and the OLE Custom Control Developer's Kit. (Tutorial)
Lang, Eric
Microsoft Systems Journal, v9, n9, p33(15)
Sept, 1994
DOCUMENT TYPE: Tutorial ISSN: 0889-9932 LANGUAGE: ENGLISH
RECORD TYPE: FULLTEXT; ABSTRACT
WORD COUNT: 9478 LINE COUNT: 00733

... in-process server DLL that supports OLE automation and visual editing as an inside-out **object** . More simply, it is a standard OLE 2 **object** that can be embedded within a **container** , in-place **activated** , and can utilize OLE automation. Being inside-out means that the **object** is usually active when visible and becomes Ul-active on a single **click** rather than a double click. As an in-process server, an OLE Control is in ...

...one or more client sites. When you want to insert a new control in the **container** , the **container** **creates** a new client site to receive it. Each client site can handle all communication between...Control Options dialog (see figure 14) contains various features that you can enable or disable. **Activate** when visible, as the name implies, allows your control to be **activated** automatically when it is visible.

Most OLE 2-aware **container** applications allow users to insert new OLE 2 **objects** by **selecting** the **object** from a standard Insert **Object** dialog. The Show in Insert **Object** dialog option in ControlWizard determines whether your control will be listed in **container** applications' Insert **Object** dialogs.

The Invisible at run-time option determines whether your control is displayed at run...the ClassWizard (see Figure 37). CSmileCtrl should be selected as the Class Name. Then I **click** on the CSmileCtrl **Object** ID, **select** the WM...

19/3,K/4 (Item 4 from file: 275)
DIALOG(R)File 275:Gale Group Computer DB(TM)

01685211 SUPPLIER NUMBER: 16005186 (USE FORMAT 7 OR 9 FOR FULL TEXT)
On Windows. (Microsoft's Chicago, OLE) (Column)
Bonner, Paul
Computer Shopper, v14, n7, p532(2)
July, 1994
DOCUMENT TYPE: Column ISSN: 0886-0556 LANGUAGE: ENGLISH
RECORD TYPE: FULLTEXT; ABSTRACT
WORD COUNT: 1333 LINE COUNT: 00097

... beginning. Eventually, they say, everything you do on your computer will involve manipulating smart OLE **objects** . To create a document, you'll open a blank **container** , then **drag** a spreadsheet **object** into it, or a word processing or **graphic** **object** , or all three. The **container** document's **onscreen** menus will change depending on which **object** you're working with, and it'll be a breeze to define links between different...

19/3,K/6 (Item 6 from file: 275)
DIALOG(R)File 275:Gale Group Computer DB(TM)

01672740 SUPPLIER NUMBER: 15043663 (USE FORMAT 7 OR 9 FOR FULL TEXT)
OLE made almost easy: creating containers and servers using MFC 2.5. (programming techniques for Microsoft's Object Linking and Embedding using Microsoft Function Class 2.5) (Tutorial)
DiLascia, Paul
Microsoft Systems Journal, v9, n4, p13(17)
April, 1994
DOCUMENT TYPE: Tutorial ISSN: 0889-9932 LANGUAGE: ENGLISH
RECORD TYPE: FULLTEXT; ABSTRACT
WORD COUNT: 9446 LINE COUNT: 00767

OLE made almost easy: creating containers and servers using MFC 2.5. (programming techniques for Microsoft's Object Linking and Embedding...
... it is to create an OLE app using MFC 2.5, here's how to **create** a simple **container** .
* Invoke the AppWizard, give your app a name.
* Press the OLE Options... button to bring...

...as a new object or from an existing file. You also get Edit Links and **Object** Edit/Show/Convert commands.
* If you insert a new **object** , the **container** app obtains the proper GUID for the **selected** **object** type and uses it to launch its server and creat a class factory and instance...

...things, it manages a list of COleClientItems, each of which represent a linked or embedded **object** in the **container** . COleClientItem implements three OLE interfaces: IOleClientSite, IAdviseSink, and IOleInPlaceSite. It manages the embedded/linked object...clicking the item:
void CContainView::On LButtonDblClk (Unit nFlags, CPoint pt) { On LButtonDown (nFlags, pt); // (**select** **object** under mouse) if (m[underscore]pSelection != NULL) {
m[underscore]pSelection->DoVerb(GetKeyState(VK[underscore]CONTROL...

...Figure 9 shows a partial listing of the OLE calls that transpire when you double- **click** an **object** in CONTAIN. It reveals just how much work is done by the framework.
What else...

...of things you'd expect. It handles mouse-hit detection (to tell when an OLE **object** has been **selected**), and maintains a selected item in the member m[underscore]pSelection. OnUpdate uses hints to...of a Server
As you might suspect, server apps are a little more complicated. A **container** can **create** a black box, then tell it what to do, but the server is the one...

...AfxOleInit, which initializes a lot of framework structures and calls OleInitialize (the same call is **generated** for **containers**). Next, InitInstance **creates** a document template. That's nothing new--but this is
pDocTemplate->SetServerInfo(ID[underscore]SCRIBTYPE...

...server "knows" how to create instances of your document class from its CLSID when the **container** invokes IClassFactory:: **CreateInstance** . To actually register the class factory, InitInstance calls COleTemplatesServer::RegisterAll.
Finally, InitInstance calls COleTemplateServer:: UpdateRegistry...did indeed get a yellow object, as in Figure 17 But when I deactivated the **object** or merely **selected** it as in Figure 18, it came out blank. The reason is that CPostitView::OnDraw...and in some respects superficial. I didn't even get a chance to mention data **objects** and **drag** -and-drop. But then, this is a survey course. Obviously, I'm enthusiastic about MFC...

19/3,K/10 (Item 10 from file: 275)
DIALOG(R)File 275:Gale Group Computer DB(TM)

01587729 SUPPLIER NUMBER: 13501994 (USE FORMAT 7 OR 9 FOR FULL TEXT)
Container yourself! (how to use the container control in OS/2 operating system) (A Guide to Developing on OS/2 Special Section) (Tutorial)
Branham, Randy
Computer Language, v10, n4, pS16(5)
April, 1993
DOCUMENT TYPE: Tutorial ISSN: 0749-2839 LANGUAGE: ENGLISH
RECORD TYPE: FULLTEXT; ABSTRACT
WORD COUNT: 2842 LINE COUNT: 00212

...ABSTRACT: operating system are a flexible and powerful tool if programmers know how to use them. **Containers** are a standard window class with styles and attributes. Styles govern **container** **object** **selection** , **object** positioning, **object** definition and **object** manipulation. Attributes govern **container** **display** , **container** style and **container** drawing. Programmers create **containers** by **selecting** styles and attributes and then completing a standard window call. Programmers may easily delete records...

...the records. When using tree view, programmers must define the new record's parent by **creating** a pointer. **Containers** may also allow for direct editing.
... standard window class that has some relatively simple attributes and styles. The styles control how **objects** within the **container** are **selected** , who positions the **objects** , whether the **objects** can be changed, and how we define the **objects** to the **container** . The attributes define how information is displayed within the **container** ; whether it has titles, separators, and other lines; and who draws the container. Starting with...

...SINGLESEL. All these precipitate how **objects** within the **container** are **selected** , but all three have very different behavior. CCS...

...51NGLESEL allows only one **object** within the **container** to be **selected** at any time. The other two allow for differing types of multiple **selection** .
Second is the read-only style. CCS...CONTAINER class, we can **create** **containers** by making a standard window call, as shown in Listing 1.
Here, we are **creating** a **container** with the control styles of CCS ...

...QUER YCNRINFO message before replacing it. With that, we've **created** a basic **container** that has a title.
Now that we have a basic container, let's put some...would leave holes where removed records were displayed. Even if we resized the window, the **objects** would stay in their current positions. **Selecting** this option (Arrange) would cause the **icon** view of the **container** to arrange itself, filling the **container** from left to right, top to bottom.
If the **icons** ever get out of order, we could realign them with this function. With CCS...giving users the capability to directly control their environment. I have briefly covered a few **container** highlights here, but this is not where it ends. Other areas such as **drag** and drop, sorting, filtering, and displaying mini- **icons** are definitely worth exploring.

```
Set     Items   Description
S1    1112744   OBJECT OR OBJECTS OR ICON OR ICONOGRAPHIC OR ICONS OR SYMB-
            OL OR SYMBOLS OR THUMBNAIL? OR (GUI OR GUIS OR GRAPHICAL()USE-
            R()INTERFACE?)(2N)(FILE? OR ONSCREEN()REPRESENTATION? OR PICT-
            URE? OR IMAGE?)
S2    8258686   CLICK? OR SELECT? OR DRAG? ? OR INDICAT? OR CHOOSE? OR PIC-
            K? OR INSTANTIAT? OR ACTIVAT?
S3     362799   WRAPPER OR WRAPPERS OR WRAPS OR WRAP OR WRAPPING OR ENVELO-
            P? OR CONTAINER?
S4       2210   S1 AND S2 AND S3
S5          3   S1(3N)S2 AND S3(2N)(CREATE? OR CREATING OR GENERAT? OR AUT-
            HORING OR AUTHORS OR WRITES OR LAUNCHES OR SPAWNS)
S6      12082   S1(2N)S2
S7          3   S4 AND S5
S8        132   S6 AND S3
S9        128   S4 AND (ONSCREEN? OR GUI OR GRAPHIC? OR DISPLAY? ? OR SCRE-
            EN? ? OR DESKTOP?)
S10       261   S5 OR S7 OR S8 OR S9
S11       197   RD (unique items)
S12        69   S11 NOT PY>1994
S13         0   S8 AND S9 AND S12
S14         3   S5 OR S7 OR S13
S15         3   RD (unique items)
S16       894   S1(2N)(S3 OR WRAPPED)
S17        12   S16 AND S9
S18         3   S17 AND S12
S19         3   S18 NOT S15
S20        65   S12 NOT (S14 OR S15 OR S17)
S21         1   S20 AND (OLE OR EMBED? OR OOP?)
S22        10   S20 AND (MODULE? OR SCRIPT? OR TOOL? OR CODE? OR MACRO)
S23        10   S21 OR S22
File   8:Ei Compendex(R) 1970-2005/Jul W2
         (c) 2005 Elsevier Eng.  Info. Inc.
File  35:Dissertation Abs Online 1861-2005/Jun
         (c) 2005 ProQuest Info&Learning
File  65:Inside Conferences 1993-2005/Jul W3
         (c) 2005 BLDSC all rts. reserv.
File   2:INSPEC 1969-2005/Jul W2
         (c) 2005 Institution of Electrical Engineers
File  94:JICST-EPlus 1985-2005/May W5
         (c)2005 Japan Science and Tech Corp(JST)
File 111:TGG Natl.Newspaper Index(SM) 1979-2005/Jul 19
         (c) 2005 The Gale Group
File   6:NTIS 1964-2005/Jul W2
         (c) 2005 NTIS, Intl Cpyrght All Rights Res
File 144:Pascal 1973-2005/Jul W2
         (c) 2005 INIST/CNRS
File  34:SciSearch(R) Cited Ref Sci 1990-2005/Jul W2
         (c) 2005 Inst for Sci Info
File  99:Wilson Appl. Sci & Tech Abs 1983-2005/Jun
         (c) 2005 The HW Wilson Co.
File  95:TEME-Technology & Management 1989-2005/Jun W2
         (c) 2005 FIZ TECHNIK
```

```
Set     Items   Description
S1    1112744   OBJECT OR OBJECTS OR ICON OR ICONOGRAPHIC OR ICONS OR SYMB-
            OL OR SYMBOLS OR THUMBNAIL? OR (GUI OR GUIS OR GRAPHICAL()USE-
            R()INTERFACE?)(2N)(FILE? OR ONSCREEN()REPRESENTATION? OR PICT-
            URE? OR IMAGE?)
S2    8258686   CLICK? OR SELECT? OR DRAG? ? OR INDICAT? OR CHOOSE? OR PIC-
            K? OR INSTANTIAT? OR ACTIVAT?
S3     711838   WRAPPER OR WRAPPERS OR WRAPS OR WRAP OR WRAPPING OR SHELL -
            OR SHELLS OR ENVELOP? OR CONTAINER?
S4       3593   S1 AND S2 AND S3
S5          5   S1(3N)S2 AND S3(2N)(CREATE? OR CREATING OR GENERAT? OR AUT-
            HORING OR AUTHORS OR WRITES OR LAUNCHES OR SPAWNS)
S6      12082   S1(2N)S2
S7          5   S4 AND S5
S8        218   S6 AND S3
S9        263   S4 AND (ONSCREEN? OR GUI OR GRAPHIC? OR DISPLAY? ? OR SCRE-
            EN? ? OR DESKTOP?)
S10       466   S5 OR S7 OR S8 OR S9
S11       346   RD (unique items)
S12       150   S11 NOT PY>1994
S13         7   S8 AND S9 AND S12
S14        12   S5 OR S7 OR S13
S15        12   RD (unique items)
S16      1392   S1(2N)(S3 OR WRAPPED)
S17        28   S16 AND S9
S18         9   S17 AND S12
S19         9   S18 NOT S15
File   8:Ei Compendex(R) 1970-2005/Jul W2
         (c) 2005 Elsevier Eng.  Info. Inc.
File  35:Dissertation Abs Online 1861-2005/Jun
         (c) 2005 ProQuest Info&Learning
File  65:Inside Conferences 1993-2005/Jul W3
         (c) 2005 BLDSC all rts. reserv.
File   2:INSPEC 1969-2005/Jul W2
         (c) 2005 Institution of Electrical Engineers
File  94:JICST-EPlus 1985-2005/May W5
         (c)2005 Japan Science and Tech Corp(JST)
File 111:TGG Natl.Newspaper Index(SM) 1979-2005/Jul 19
         (c) 2005 The Gale Group
File   6:NTIS 1964-2005/Jul W2
         (c) 2005 NTIS, Intl Cpyrght All Rights Res
File 144:Pascal 1973-2005/Jul W2
         (c) 2005 INIST/CNRS
File  34:SciSearch(R) Cited Ref Sci 1990-2005/Jul W2
         (c) 2005 Inst for Sci Info
File  99:Wilson Appl. Sci & Tech Abs 1983-2005/Jun
         (c) 2005 The HW Wilson Co.
File  95:TEME-Technology & Management 1989-2005/Jun W2
         (c) 2005 FIZ TECHNIK
```

15/5/11 (Item 1 from file: 6)
DIALOG(R)File 6:NTIS

1694903 NTIS Accession Number: AD-A257 429/1
UT20-PCTE Browser Tool Version Description Document Version 0.1
(Informal technical data)
Herton, M. J.
Paramax Systems Corp., Reston, VA. Tactical Systems Div.
Corp. Source Codes: 103391001; 424460
12 Jun 92 25p
Languages: English
Journal Announcement: GRAI9305
Order this product from NTIS by: phone at 1-800-553-NTIS (U.S. customers); (703)605-6000 (other countries); fax at (703)321-8547; and email at orders@ntis.fedworld.gov. NTIS is located at 5285 Port Royal Road, Springfield, VA, 22161, USA.
NTIS Prices: PC A03/MF A01
Country of Publication: United States
Contract No.: F19628-88-D-0031
The PCTE Browser Tool (PBT) is designed to **graphically display** parts of a PCTE **object** base. **Selected objects** in the **object** base and the relationships amongst these **objects** are displayed at the PBT user's request. The PBT is intended to complement text-oriented commands such as obj-list-links and obj-list-attr that are included with the Emeraude PCTE 1.5 release-commands intended to be invoked from the text-oriented esh command **shell** . PBT version 0.1 is an alpha release of the browser.
Descriptors: *Computer program documentation; *Data **displays** ; Release; Computer files; Feedback
Identifiers: *Software tools; NTISDODXA
Section Headings: 62B (Computers, Control, and Information Theory--Computer Software)

```
Set     Items   Description
S1     882355   OBJECT OR OBJECTS OR ICON OR ICONOGRAPHIC OR ICONS OR SYMB-
             OL OR SYMBOLS OR THUMBNAIL? OR (GUI OR GUIS OR GRAPHICAL()USE-
             R()INTERFACE?)(2N)(FILE? OR ONSCREEN()REPRESENTATION? OR PICT-
             URE? OR IMAGE?)
S2    1347083   CLICK? OR SELECT? OR DRAG? ? OR INDICAT? OR CHOOSE? OR PIC-
             K? OR INSTANTIAT? OR ACTIVAT?
S3     435367   WRAPPER OR WRAPPERS OR WRAPS OR WRAP OR WRAPPING OR SHELL -
             OR SHELLS OR ENVELOP? OR CONTAINER?
S4        961   S1(5N)S2(5N)S3
S5       6472   S3(2N)(START OR STARTS OR STARTING OR LAUNCH? OR BEGIN OR -
             AUTHOR OR AUTHORING OR CREATE? OR CREATING OR INITIAT? OR SPA-
             WN?)
S6         46   S4(10N)S5
S7       1244   S1(2N)S2(10N)S3
S8        177   S7 AND IC=G06F-009
S9         36   S6 AND IC=G06F
S10       196   S8 OR S9
S11       150   S10 NOT AD=19941115:19971115
S12        95   S11 NOT AD=19971115:20001115
S13        54   S12 NOT AD=20001115:20031115
S14        54   S13 NOT AD=20031115:20050801
S15        53   S14 AND IC=G06F-009
S16        17   S15(10N)(ONSCREEN? OR ON()SCREEN OR GUI OR GUIS OR DESKTOP?
              OR DESK()(TOP OR TOPS) OR USER()INTERFACE? OR DISPLAY OR GRA-
             PHIC?()(OBJECT OR OBJECTS))
File 348:EUROPEAN PATENTS 1978-2005/Jul W02
         (c) 2005 European Patent Office
File 349:PCT FULLTEXT 1979-2005/UB=20050714,UT=20050707
         (c) 2005 WIPO/Univentio
```

16/3,K/1 (Item 1 from file: 348)
DIALOG(R)File 348:EUROPEAN PATENTS

00897620
A method and system for in-place interaction with embedded objects
Verfahren und System fur die In-Ort-Wechselwirkung mit eingebetteten Objekten
Procede et systeme qui permet l'interaction-en-place avec des objets encastres
PATENT ASSIGNEE:
MICROSOFT CORPORATION, (749861), One Microsoft Way, Redmond, Washington 98052-6399, (US), (applicant designated states: DE;FR;GB)
INVENTOR:
Koppolu, Scrinivasa R., 2402 236th Avenue N.E., Redmind, Washington 98052, (US)
MacKichan, Barry B., 12730 Manzanita Road N.E., Bainbridge Island, Washington 98110, (US)
McDaniel, Richard, 6017 Stanton Avenue, Apt 1, Pittsburgh, Pennsylvania 15206, (US)
Remala, Rao V., 19011 N.E. 151st Street, Woodinville, Washington 98072, (US)
Williams, Antony S., 7800 S.E 22nd Place, Mercer Island ,Washington 98040, (US)
LEGAL REPRESENTATIVE:
Grunecker, Kinkeldey, Stockmair & Schwanhausser Anwaltssozietat (100721), Maximilianstrasse 58, 80538 Munchen, (DE)
PATENT (CC, No, Kind, Date): EP 820008 A2 980121 (Basic)
APPLICATION (CC, No, Date): EP 97117414 931124;
PRIORITY (CC, No, Date): US 984868 921201
DESIGNATED STATES: DE; FR; GB
RELATED PARENT NUMBER(S) - PN (AN):
EP 672277 (EP 949024079)
INTERNATIONAL PATENT CLASS: G06F-009/44; G06F-009/46; G06F-003/033;
ABSTRACT WORD COUNT: 128

LANGUAGE (Publication,Procedural,Application): English; English; English
FULLTEXT AVAILABILITY:

Available Text	Language	Update	Word Count
CLAIMS A	(English)	9804	687
SPEC A	(English)	9804	16912
Total word count - document A			17599
Total word count - document B			0
Total word count - documents A + B			17599

...SPECIFICATION server application with a server window environment with server resources for interacting with the containee **object** . The method of the present invention displays the **container** window environment on a **display** device. A user then **selects** the containee **object** . In response to **selecting** the containee object, the method integrates a plurality of the server resources with the displayed...

...or embedded object.
Figure 7 is a block diagram showing a public view of an **object** .
Figure 8 is a sample user menu provided by a **container** application to **display** and **select** the actions available for an **object** .
Figure 9 is a diagram showing the composite menu bar resulting from the merger of...a menu.
Figure 8 is a sample user menu provided by a container application to **display** and **select** the actions available for an **object** . Menu item 803 is the entry for the **object** on the **container** application Edit menu 802. The entry varies based on the currently **selected object** . When no embedded or linked objects are **selected** , menu item 803 is not displayed. Submenu 804 displays the actions supported by an "Excel... performed is setting a flag in step 1701 indicating that a containee

object has been **activated** . This information is used later, whenever the specified **object** 's parent **container object** is asked to **activate** or deactivate. This flag tells the parent **container** application whether it needs to **activate** or deactivate an **object** contained within it (a nested object), instead of **activating** or deactivating its own **user interface** .

16/3,K/3 (Item 3 from file: 348)
DIALOG(R)File 348:EUROPEAN PATENTS

00672307
1GRAPHICAL USER INTERFACE **AND METHOD TO** SELECTIVELY **OPEN** CONTAINER OBJECTS **DURING** DRAG **AND DROP OPERATIONS**
GRAPHISCHE BENUTZEROBERFLACHE UND VERFAHREN ZUM SELEKTIVEN OFFNEN VON CONTAINEROBJEKTEN WAHREND VERSCHIEBUNGS- UND ABLAGERUNGSOPERATIONEN
INTERFACE UTILISATEUR GRAPHIQUE ET PROCEDE D'OUVERTURE SELECTIVE D'ICONES CONTENANT DES OBJETS PENDANT LES OPERATIONS DE TRAINEE ET DE LACHAGE
PATENT ASSIGNEE:
APPLE COMPUTER, INC., (1211950), 20525 Mariani Avenue, Cupertino, California 95014, (US), (Proprietor designated states: all)
INVENTOR:
CONRAD, Thomas, J., 1240 San Tomas Aquino #205,, San Jose, CA 95117, (US)
WONG, Yin, Yin, 970 Alice Lane #4,, Menlo Park, CA 94025, (US)
LEGAL REPRESENTATIVE:
Wombwell, Francis (46021), Potts, Kerr & Co. 15, Hamilton Square, Birkenhead Merseyside L41 6BR, (GB)
PATENT (CC, No, Kind, Date): EP 702811 A1 960327 (Basic)
EP 702811 B1 000223
WO 9429787 941222
APPLICATION (CC, No, Date): EP 94920069 940603; WO 94US6241 940603
PRIORITY (CC, No, Date): US 76253 930611
DESIGNATED STATES: DE; FR; GB
INTERNATIONAL PATENT CLASS: G06F-003/033; G06F-009/44
NOTE:
No A-document published by EPO
LANGUAGE (Publication,Procedural,Application): English; English; English
FULLTEXT AVAILABILITY:

Available Text	Language	Update	Word Count
CLAIMS B	(English)	200008	2528
CLAIMS B	(German)	200008	2571
CLAIMS B	(French)	200008	2763
SPEC B	(English)	200008	6598
Total word count - document A			0
Total word count - document B			14460
Total word count - documents A + B			14460

GRAPHICAL USER INTERFACE **AND METHOD TO** SELECTIVELY **OPEN** CONTAINER OBJECTS **DURING** DRAG **AND DROP OPERATIONS**

16/3,K/4 (Item 4 from file: 348)
DIALOG(R)File 348:EUROPEAN PATENTS

00667233
Method and system for presenting contents of a container object within a graphical user interface in a data processing system.
Verfahren und System zur Anzeige des Inhalts eines Behalterdatenobjektes in einer graphischen Benutzerschnittstelle in einem Datenverarbeitungssyst em.
Procede et systeme pour la visualisation des contenus d'un objet recipient dans une interface utilisateur graphique d'un systeme de traitement de donnees.
PATENT ASSIGNEE:
International Business Machines Corporation, (200120), Old Orchard Road, Armonk, N.Y. 10504, (US), (applicant designated states: DE;FR;GB)
INVENTOR:
Henshaw, Susan F., 106 Old Rock Hampton Lane, Cary Wake County, North CArolina 27513, (US)
Redpath, Sarah D., 109 Blythewood Court, Cary Wake County, North Carolina 27513, (US)
LEGAL REPRESENTATIVE:
de Pena, Alain et al (15151), Compagnie IBM France Departement de Propriete Intellectuelle, 06610 La Gaude, (FR)
PATENT (CC, No, Kind, Date): EP 640915 A1 950301 (Basic)
APPLICATION (CC, No, Date): EP 94480060 940707;
PRIORITY (CC, No, Date): US 113557 930827
DESIGNATED STATES: DE; FR; GB
INTERNATIONAL PATENT CLASS: G06F-009/44; G06F-003/033;
ABSTRACT WORD COUNT: 114

LANGUAGE (Publication,Procedural,Application): English; English; English
FULLTEXT AVAILABILITY:
Available Text Language Update Word Count
CLAIMS A (English) EPAB95 720
SPEC A (English) EPAB95 2599
Total word count - document A 3319
Total word count - document B 0
Total word count - documents A + B 3319

...SPECIFICATION a user. Such "folders" typically require a user to perform some action in order to **select** , locate and "store" **objects** within the "folder." Some known **container objects** may also be manipulated to perform user specified activities utilizing the **objects** stored within the **container object** . For example, a user may specify that all **objects** stored within **selected container object** be printed.
A graphical **user interface** may **display** multiple **icons** which represent **objects** . These icons may be displayed in any order throughout the graphical user interface. Often these...

...user interface in order to help distinguish the window from the rest of the graphical **user interface** .
When a **container object** is **selected** and opened by a user, the contents of the **container object** are typically displayed within a window. In this manner, a user may distinguish the contents of the **selected container object** from other **objects** that might be currently displayed within the graphical **user interface** .
It frequently becomes necessary to **select** and **display** the contents of multiple **container objects** simultaneously. To do this, multiple windows are created, each displaying the contents of one of the **selected container objects** . Due to the limited size of a computer **display** device, such as a display screen, in order to display multiple windows, the windows must...passes to block 88 which depicts a definition of a display space at the determined **display** screen coordinates.
Block 90 illustrates the displaying of an **icon** representing the

selected container object in the upper left corner of the **display** space. Thereafter, the process passes to block 92 which illustrates the presentation of all contents of the **selected container object** in the **display** space to the right of the **icon** utilizing the specified grid pattern. The process then terminates as depicted at block 94.

Referring...

...block 116.

Block 118 illustrates the decreasing of the intensity of each pixel in the **display** space. Next, block 120 depicts the displaying of an **icon** representing the **selected container object** in the upper left corner of the **display** space. Block 122 then illustrates the presentation of the contents of the **selected container object** in the **display** space to the right of the **icon** utilizing a specified grid pattern. The process then passes to block 124 which depicts the...

16/3,K/8 (Item 8 from file: 348)
DIALOG(R)File 348:EUROPEAN PATENTS
(c) 2005 European Patent Office. All rts. reserv.

00551324
Container object management system
System zur Verwaltung von "Behalter" Datenobjekten
Systeme de gestion d'objets "recipients"
PATENT ASSIGNEE:
International Business Machines Corporation, (200120), Old Orchard Road, Armonk, N.Y. 10504, (US), (Proprietor designated states: all)
INVENTOR:
Miller, Patrice R., 986 Quail Ridge, Keller, Texas 76248, (US)
Miller, Wade A., 986 Quail Ridge, Keller, Texas 76248, (US)
Rayborn, John K., 5631 Southwestern Boulevard, Dallas, Texas 75209, (US)
LEGAL REPRESENTATIVE:
de Pena, Alain (15151), Compagnie IBM France Departement de Propriete Intellectuelle, 06610 La Gaude, (FR)
PATENT (CC, No, Kind, Date): EP 520924 A2 921230 (Basic)
EP 520924 A3 930908
EP 520924 B1 991117
APPLICATION (CC, No, Date): EP 92480078 920603;
PRIORITY (CC, No, Date): US 723086 910628
DESIGNATED STATES: DE; FR; GB
INTERNATIONAL PATENT CLASS: G06F-009/44
ABSTRACT WORD COUNT: 239
NOTE:
Figure number on first page: 3

LANGUAGE (Publication,Procedural,Application): English; English; English
FULLTEXT AVAILABILITY:

Available Text	Language	Update	Word Count
CLAIMS B	(English)	9946	426
CLAIMS B	(German)	9946	472
CLAIMS B	(French)	9946	491
SPEC B	(English)	9946	5246
Total word count - document A			0
Total word count - document B			6635
Total word count - documents A + B			6635

...SPECIFICATION AddObject - adds an object to the container pane.
2. ContainerCopy - ability to copy a contained **object** to another **container** .
3. **ContainerMove** - ability to move a contained **object** to another **container** .
4. DefaultDetailsInfo - default code to **display** contained **objects** in detail format.
5. **Drag** - handles an object being dragged within the pane or over the pane.
6. Drop - handles...

...on the pane.
7. ObjectDoubleClicked - handles a contained object being doubleclicked.
8. Refresh - refresh the **display** of contained **objects** .
9. Remove - remove a contained **object** from the **container** pane.
10. Rename - rename a contained **object** to a new name.
Drag involves direct manipulation of objects. In OfficeVision/2 using mouse button 2, an object can...container manager. The process then proceeds to block 146, showing container manager determining which contained **object** is to be dragged. Next, **container** manager queries contained **object** for an **icon** to use during **drag** as illustrated in block 148. Thereafter, block 150 shows **container** manager building a data structure to enable the **GUI** operating environment to handle the drag. The data structure is a C structure that can...

16/3,K/9 (Item 9 from file: 348)
DIALOG(R)File 348:EUROPEAN PATENTS

00511447
COMPUTER LOG-ON
COMPUTER LOG-ON
PROCEDURE D'ENTREE DANS UN SYSTEME INFORMATIQUE
PATENT ASSIGNEE:
Hewlett-Packard Company, (206030), 3000 Hanover Street, Palo Alto, California 94304, (US), (applicant designated states: DE;FR;GB)
INVENTOR:
MOREL, William, c/o Hewlett-Packard Limited, Filton Road, Stoke Gifford, Bristol BS12 6QZ, (GB)
DUGGAN, Hugh, c/o Hewlett-Packard Limited, Filton Road, Stoke Gifford, Bristol BS12 6QZ, (GB)
LEGAL REPRESENTATIVE:
Squibbs, Robert Francis et al (36277), Intellectual Property Section Building 2 Hewlett-Packard Limited Filton Road, Stoke Gifford Bristol BS12 6QZ, (GB)
PATENT (CC, No, Kind, Date): EP 527213 A1 930217 (Basic)
EP 527213 B1 980603
WO 9117502 911114
APPLICATION (CC, No, Date): EP 91920960 910430; WO 91GB688 910430
PRIORITY (CC, No, Date): GB 9009703 900430
DESIGNATED STATES: DE; FR; GB
INTERNATIONAL PATENT CLASS: G06F-009/445;
NOTE:
No A-document published by EPO
LANGUAGE (Publication,Procedural,Application): English; English; English
FULLTEXT AVAILABILITY:

Available Text	Language	Update	Word Count
CLAIMS B	(English)	9823	400
CLAIMS B	(German)	9823	372
CLAIMS B	(French)	9823	441
SPEC B	(English)	9823	2131
Total word count - document A			0
Total word count - document B			3344
Total word count - documents A + B			3344

...SPECIFICATION an OBJECT MANAGER which is an application running in the windows environment and which controls **activation** and deactivation of **objects** , and the passage of messages between **objects** .
An **object** is sometimes a **CONTAINER** which contains as notional parts other **objects** . Examples of **container objects** in a distributed office system are a **desktop** , folder and a document. A VARIABLE DIMENSION DATA OBJECT (VDO) is a data store of...

16/3,K/10 (Item 10 from file: 348)
DIALOG(R)File 348:EUROPEAN PATENTS

00504097
OBJECT BASED COMPUTER SYSTEM
OBJEKTBASIERTES RECHNERSYSTEM
SYSTEME INFORMATIQUE ORIENTE OBJETS
PATENT ASSIGNEE:
Hewlett-Packard Company, (206030), 3000 Hanover Street, Palo Alto, California 94304, (US), (applicant designated states: DE;FR;GB)
INVENTOR:
DUGGAN, Hugh Hewlett-Packard Limited Filton Road, Stoke Gifford, Bristol BS12 6QZ, (GB)
MOREL, William, Paul Hewlett-Packard Limited, Filton Road Stoke Gifford, Bristol BS12 6QZ, (GB)
Robson, Christopher Hewlett-Packard Ltd., Felton Road, Stoke Gifford Bristol BS12 6QZ, (GB)
LEGAL REPRESENTATIVE:
Squibbs, Robert Francis et al (36277), Intellectual Property Section Building 2 Hewlett-Packard Limited Filton Road, Stoke Gifford Bristol BS12 6QZ, (GB)
PATENT (CC, No, Kind, Date): EP 527833 A1 930224 (Basic)
EP 527833 B1 980603
WO 9117499 911114
APPLICATION (CC, No, Date): EP 91908881 910430; WO 91GB690 910430
PRIORITY (CC, No, Date): GB 9009699 900430
DESIGNATED STATES: DE; FR; GB
INTERNATIONAL PATENT CLASS: G06F-009/44;
NOTE:
No A-document published by EPO
LANGUAGE (Publication,Procedural,Application): English; English; English
FULLTEXT AVAILABILITY:

Available Text	Language	Update	Word Count
CLAIMS B	(English)	9823	413
CLAIMS B	(German)	9823	406
CLAIMS B	(French)	9823	472
SPEC B	(English)	9823	3587
Total word count - document A			0
Total word count - document B			4878
Total word count - documents A + B			4878

...SPECIFICATION an OBJECT MANAGER which is an application running in the windows environment and which controls **activation** and deactivation of **objects** , and the passage of messages between **objects** .
An **object** is sometimes a **CONTAINER** which contains as notional parts other **objects** . Examples of **container objects** in a distributed office system are a **desktop** , folder and a document. A VARIABLE DIMENSION DATA OBJECT (VDO) is a data store of...

```
Set     Items   Description
S1     577326   OBJECT OR OBJECTS OR ICON OR ICONOGRAPHIC OR ICONS OR SYMB-
             OL OR SYMBOLS OR THUMBNAIL? OR (GUI OR GUIS OR GRAPHICAL()USE-
             R()INTERFACE?)(2N)(FILE? OR ONSCREEN()REPRESENTATION? OR PICT-
             URE? OR IMAGE?)
S2    2677128   CLICK? OR SELECT? OR DRAG? ? OR INDICAT? OR CHOOSE? OR PIC-
             K? OR INSTANTIAT? OR ACTIVAT?
S3     874974   WRAPPER OR WRAPPERS OR WRAPS OR WRAP OR WRAPPING OR SHELL -
             OR SHELLS OR ENVELOP? OR CONTAINER?
S4        486   S1(3N)S2 AND S3
S5       1782   S3(2N)(START OR LAUNCH OR BEGIN OR AUTHOR OR AUTHORING OR -
             CREATE? OR CREATING OR INITIAT? OR SPAWN?)
S6          8   S4 AND S5
S7        114   S4 AND IC=G06F
S8        118   S6 OR S7
S9         70   S8 NOT AD=19941115:19971115
S10        55   S9 NOT AD=19971115:20001115
S11        29   S10 NOT AD=20001115:20031115
S12        28   S11 NOT AD=20031115:20050901
File 347:JAPIO Nov 1976-2005/Feb(Updated 050606)
         (c) 2005 JPO & JAPIO
File 350:Derwent WPIX 1963-2005/UD,UM &UP=200545
         (c) 2005  Thomson Derwent
```

12/5/2 (Item 2 from file: 347)
DIALOG(R)File 347:JAPIO

03020550 **Image available**
PARALLEL PROCESSING TYPE DOCUMENT PROCESSOR

PUB. NO.: 01-318150 [JP 1318150 A]
PUBLISHED: December 22, 1989 (19891222)
INVENTOR(s): ANDO MAKOTO
APPLICANT(s): FUJI XEROX CO LTD [359761] (A Japanese Company or Corporation), JP (Japan)
APPL. NO.: 63-151462 [JP 88151462]
FILED: June 20, 1988 (19880620)
INTL CLASS: [4] **G06F-015/20**
JAPIO CLASS: 45.4 (INFORMATION PROCESSING -- Computer Applications)
JOURNAL: Section: P, Section No. 1018, Vol. 14, No. 124, Pg. 64, March 08, 1990 (19900308)

ABSTRACT

PURPOSE: To easily detect whether a function **indicated** by an **icon** is executing background processing or not by switching an icon transferred to the execution of background processing to display with a format different from that of other icons.

CONSTITUTION: A display control part 5 checks whether an icon newly started the background processing exists in a disk top window or not, and when the icon exists, switches the icon to another pattern (ghost icon) in order to inform the background processing of the icon. When the icon does not exist in the disk top window, but a **container** window is opened, an icon in the **container** window is switched to a ghost icon shown by another pattern.

12/5/3 (Item 1 from file: 350)
DIALOG(R)File 350:Derwent WPIX

011214467
WPI Acc No: 1997-192392/199717
Related WPI Acc No: 1994-218083; 1997-033851; 1997-297648; 1998-311631; 1998-494995
XRPX Acc No: N97-158997

Interacting with containee object within container **object for linked and embedded objects for spreadsheet application - determining whether to invoke** container **application code or server application code and executing server application code as separately scheduled entity to process server resource selection when selected resource is server resource**

Patent Assignee: MICROSOFT CORP (MICR-N)
Inventor: HODGES C D; KOPPOLU S R; MACKICHAN B B; MCDANIEL R; REMALA R V; WILLIAMS A S
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
US 5613058	A	19970318	US 92984868	A	19921201	199717	B
			US 94229264	A	19940415		

Priority Applications (No Type Date): US 94229264 A 19940415; US 92984868 A 19921201
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
US 5613058	A		97	G06F-015/00	CIP of application US 92984868

Abstract (Basic): US 5613058 A

The **container** object has a **container** application with a **container** window environment that has **container** resources for interacting with the **container** object. The containee object has a server application with a server window environment with server resources for interacting with the containee object.

The **container** window environment is displayed on a display. A user then **selects** the containee **object** . In response to **selecting** the containee **object** , the server resources are integrated with the displayed **container** window environment. When a user then selects a server resource, the server application is invoked to process the server resource selection. Conversely, when a user selects a **container** resource, the **container** application is invoked to process the **container** resource selection.

Title Terms: INTERACT; OBJECT; **CONTAINER** ; OBJECT; LINK; EMBED; OBJECT; APPLY; DETERMINE; INVOKE; **CONTAINER** ; APPLY; CODE; SERVE; APPLY; CODE; EXECUTE; SERVE; APPLY; CODE; SEPARATE; SCHEDULE; ENTITY; PROCESS; SERVE; RESOURCE; SELECT; SELECT; RESOURCE; SERVE; RESOURCE
Derwent Class: T01
International Patent Class (Main): **G06F-015/00**
International Patent Class (Additional): **G06F-001/00**
File Segment: EPI

12/5/4 (Item 2 from file: 350)
DIALOG(R)File 350:Derwent WPIX

010368396
WPI Acc No: 1995-269757/199536
XRPX Acc No: N95-207491
Object-oriented workplace application program design tool - has set of class templates represented on workarea by icons allowing numerous constructional functions to be performed by user
Patent Assignee: LIU E K C (LIUE-I)
Inventor: HO A Y; LIU E K C
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
CA 2110970	A	19950609	CA 2110970	A	19931208	199536	B

Priority Applications (No Type Date): CA 2110970 A 19931208
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
CA 2110970	A		102	G06F-015/403	

Abstract (Basic): CA 2110970 A
The tool includes a set of predefined class templates to effect different functional utilities in an applications program. The class templates have predefined relationships with other templates allowing template interaction. Each template is represented on a computer screen by an icon.
A workarea provides a substrate for building an applications program. A mouse enables a user to select, pick-up, **drag** and drop **icons** from one place to another on a computer screen workplace containing the workarea. The class templates include a data class, a query function class, a form class, a function class, a graphics class, a data **container** class, a static **container** class, a data file class and a database class.
ADVANTAGE - Avoids requirement for tedious, error prone and time consuming programming.
Dwg.0/0
Title Terms: OBJECT; ORIENT; APPLY; PROGRAM; DESIGN; TOOL; SET; CLASS; TEMPLATE; REPRESENT; ALLOW; NUMEROUS; CONSTRUCTION; FUNCTION; PERFORMANCE ; USER
Derwent Class: T01
International Patent Class (Main): **G06F-015/403**
File Segment: EPI

12/5/6 (Item 4 from file: 350)
DIALOG(R)File 350:Derwent WPIX

010300812 **Image available**
WPI Acc No: 1995-202072/199527
XRPX Acc No: N95-158733
Automatic storage method for object in GUI - involves displaying iconic representation of container object indicating storage of objects associated with it allowing organisation of objects
Patent Assignee: INT BUSINESS MACHINES CORP (IBMC); IBM CORP (IBMC)
Inventor: HENSHAW S F; REDPATH S D
Number of Countries: 005 Number of Patents: 005
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
EP 656580	A2	19950607	EP 94480120	A	19941108	199527	B
JP 7200244	A	19950804	JP 94226208	A	19940921	199540	
US 5619637	A	19970408	US 93160623	A	19931202	199720	
EP 656580	B1	20020123	EP 94480120	A	19941108	200207	
DE 69429711	E	20020314	DE 629711	A	19941108	200226	
			EP 94480120	A	19941108		

Priority Applications (No Type Date): US 93160623 A 19931202
Cited Patents: No-SR.Pub
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
EP 656580	A2	E	14	G06F-003/033	
Designated States (Regional): DE FR GB					
JP 7200244	A		9	G06F-003/14	
US 5619637	A		13	G06F-019/00	
EP 656580	B1	E		G06F-003/033	
Designated States (Regional): DE FR GB					
DE 69429711	E			G06F-003/033	Based on patent EP 656580

Abstract (Basic): EP 656580 A
The method involves specifying an object within a graphical user interface, specifying a **container** object with the graphical user interface, and associating the object with the **container** object.
The object is removed from the **container** object for use in the data processing system. The object is automatically stored within the **container** object in response to the completion of the use of the object, this provides an organisational enhancement for the objects within the **container** object.
ADVANTAGE - Provides automatic storage of object within **container** object within GUI in data processing system.
Dwg.5/8
Title Terms: AUTOMATIC; STORAGE; METHOD; OBJECT; DISPLAY; REPRESENT; **CONTAINER** ; OBJECT; INDICATE; STORAGE; OBJECT; ASSOCIATE; ALLOW; ORGANISE ; OBJECT
Derwent Class: T01
International Patent Class (Main): **G06F-003/033** ; **G06F-003/14** ; **G06F-019/00**
International Patent Class (Additional): **G06F-009/44** ; G06T-001/00
File Segment: EPI

12/5/10 (Item 8 from file: 350)
DIALOG(R)File 350:Derwent WPIX

009950370 **Image available**
WPI Acc No: 1994-218083/199426
Related WPI Acc No: 1997-033851; 1997-192392; 1997-297648; 1998-311631; 1998-494995
XRPX Acc No: N94-172175

In-place interaction with computer embedded objects - editing and otherwise interacting with contained object within context of application window environment, combining menus of object server application

Patent Assignee: MICROSOFT CORP (MICR-N)
Inventor: KOPPOLU S R; MACKICHAN B B; MCDANIEL R; REMALA R V; WILLIAMS A S
Number of Countries: 018 Number of Patents: 008
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week
WO 9414115	A2	19940623	WO 93US11468	A	19931124	199426 B
WO 9414115	A3	19940929	WO 93US11468	A	19931124	199518
EP 672277	A1	19950920	WO 93US11468	A	19931124	199542
			EP 94902407	A	19931124	
JP 8500200	W	19960109	WO 93US11468	A	19931124	199642
			JP 94514206	A	19931124	
EP 820008	A2	19980121	EP 94902407	A	19931124	199808
			EP 97117414	A	19931124	
EP 672277	B1	19980513	WO 93US11468	A	19931124	199823
			EP 94902407	A	19931124	
			EP 97117414	A	19931124	
DE 69318571	E	19980618	DE 618571	A	19931124	199830
			WO 93US11468	A	19931124	
			EP 94902407	A	19931124	
JP 3181592	B2	20010703	WO 93US11468	A	19931124	200139
			JP 94514206	A	19931124	

Priority Applications (No Type Date): US 92984868 A 19921201
Cited Patents: 3.Jnl.Ref; EP 215203; EP 304071; No-SR.Pub
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
WO 9414115	A2	E	91	G06F-009/44	
Designated States (National): CA JP					
Designated States (Regional): AT BE CH DE DK ES FR GB GR IE IT LU MC NL PT SE					
WO 9414115	A3			G06F-009/44	
EP 672277	A1	E	91	G06F-009/44	Based on patent WO 9414115
Designated States (Regional): DE FR GB					
JP 8500200	W		126	G06F-017/21	Based on patent WO 9414115
EP 820008	A2	E	64	G06F-009/44	Div ex application EP 94902407 Div ex patent EP 672277
Designated States (Regional): DE FR GB					
EP 672277	B1	E	71	G06F-003/033	Related to application EP 97117414 Related to patent EP 820008 Based on patent WO 9414115
Designated States (Regional): DE FR GB					
DE 69318571	E			G06F-003/033	Based on patent EP 672277 Based on patent WO 9414115
JP 3181592	B2		63	G06F-017/21	Previous Publ. patent JP 8500200 Based on patent WO 9414115

Abstract (Basic): WO 9414115 A

The method requires a user to **select** the **container object** and integrate several server responses with the window environment. The server application processes the selection. When a user selects a **container** resource, the application processes the **container** resource selection.

The **container** application has **container** menus and also the

server where the steps of integrating several resources generates a composite menu and the integrating interleaves server and **container** menus in the composite menu bar.

USE - Interaction with linked and embedded objects within the context of its **container** application for document processing for compound documents.

Dwg.9/31

Title Terms: PLACE; INTERACT; COMPUTER; EMBED; OBJECT; EDIT; INTERACT; CONTAIN; OBJECT; CONTEXT; APPLY; WINDOW; ENVIRONMENT; COMBINATION; MENU; OBJECT; SERVE; APPLY

Derwent Class: T01

International Patent Class (Main): **G06F-003/033** ; **G06F-009/44** ; **G06F-017/21**

International Patent Class (Additional): **G06F-003/14** ; **G06F-009/06** ; **G06F-009/46**

File Segment: EPI

12/5/12 (Item 10 from file: 350)
DIALOG(R)File 350:Derwent WPIX

009479845
WPI Acc No: 1993-173380/199321
XRPX Acc No: N93-132941
Providing virtual classes for discrete objects without sub-classing - enabling user or developer to easily modify folder or any other object to gain locking or timed property
Patent Assignee: ANONYMOUS (ANON)
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
RD 348077	A	19930410	RD 93348077	A	19930320	199321	B

Priority Applications (No Type Date): RD 93348077 A 19930320
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
RD 348077	A		1	G06F-000/00	

Abstract (Basic): RD 348077 A

A Virtual Class is created called VC password. The class provides an **enveloping** mechanism which can be absorbed into existing class based objects. Virtual classes cannot exist as an instantiation, it must **envelope** an existing object or class. An **envelope** template is provided which a user can **drag** to an existing **Object** to inherit the characteristics of the Virtual Class.

To create a WP Folder Class object with a password characteristics the user can create a Folder Object using the WP Folder Class template then drag the VC password **envelope** template onto the Object. The object is now a password protected folder object. In addition, the user could drag the VC password template onto an existing Class Template to create a template which will produce class objects with the VC password inherited characteristic.

ADVANTAGE - Reduction of production of class names to facilitate characteristics from superclasses.

Dwg.0/0

Title Terms: VIRTUAL; CLASS; DISCRETE; OBJECT; SUB; ENABLE; USER; DEVELOP; EASY; MODIFIED; FOLDER; OBJECT; GAIN; LOCK; TIME; PROPERTIES
Derwent Class: T01
International Patent Class (Main): **G06F-000/00**
File Segment: EPI

12/5/14 (Item 12 from file: 350)
DIALOG(R)File 350:Derwent WPIX

009413953
WPI Acc No: 1993-107464/199313
XRPX Acc No: N93-081751
Action- object **availability** indication **- allowing users to determine actions available for** objects **already** selected **, actions available for action-** object selection **and actions unavailable to either**
Patent Assignee: ANONYMOUS (ANON)
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
RD 346092	A	19930210	RD 93346092	A	19930120	199313	B

Priority Applications (No Type Date): RD 93346092 A 19930120
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
RD 346092	A		1	G06F-000/00	

Abstract (Basic): RD 346092 A

The technique allows the user to choose to display a pull-down menu without first **selecting** an **object** , actions that require an object would appear grayed. Actions that are available for action- **object selection** would appear yellow. If the user **selects** an **object** first and then **selects** a pull-down menu, actions available for the **selected object** would appear green, actions unavailable for the **selected object** yet available for an action-object sequence would appear yellow and actions unavailable for an action-object sequence yet also unavailable for the **selected object** would appear grayed.

The technique identifies actions that never require **object selection** , such as those that have their effect on a **container** of objects. Examples include Sort, Find or Refresh. An additional colour could be used to identify these type of actions or the green colour could be used to **indicate** that no further **object selection** is necessary.

ADVANTAGE - Allows users to determine actions that are available for action-object sequences that might otherwise be grayed out.

Dwg.0/0

Title Terms: ACTION; OBJECT; AVAILABLE; INDICATE; ALLOW; USER; DETERMINE; ACTION; AVAILABLE; OBJECT; SELECT; ACTION; AVAILABLE; ACTION; OBJECT; SELECT; ACTION; UNAVAILABLE
Derwent Class: T01
International Patent Class (Main): **G06F-000/00**
File Segment: EPI

12/5/19 (Item 17 from file: 350)
DIALOG(R)File 350:Derwent WPIX

009178493
WPI Acc No: 1992-305928/199237
XRPX Acc No: N92-234070
Dynamically changing appearance of mouse pointer - using colour change and arrow pointing to object to provide greater feedback to user
Patent Assignee: ANONYMOUS (ANON)
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
RD 340038	A	19920810	RD 92340038	A	19920720	199237	B

Priority Applications (No Type Date): RD 92340038 A 19920720
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
RD 340038	A		1	G06F-000/00	

Abstract (Basic): RD 340038 A
Two appearance changes are specified: colour change and directional movement. The colour of the mouse pointer changes from white to blue or from red to green, as the user moves the pointer from a non-selectable to a selectable area of the window. This change provides a subtle cue to the user that he is in a selectable area of the screen, e.g. without having to point right on top of it.
When the pointer is moved into a selectable area, the arrow points directly towards the centre of the **selectable object** . The arrow rotates a full 360 deg. to remain pointing to the object as the pointer is moved about the object. This technique is most useful when selectable areas are spaced closely together, such as icons on a desktop **container** .
USE -Changing appearance of mouse pointer as it is moved around window.
Dwg.0/0
Title Terms: DYNAMIC; CHANGE; APPEAR; MOUSE; POINT; COLOUR; CHANGE; ARROW; POINT; OBJECT; GREATER; FEEDBACK; USER
Derwent Class: T04
International Patent Class (Main): **G06F-000/00**
File Segment: EPI

```
Set     Items   Description
S1     577308   OBJECT OR OBJECTS OR ICON OR ICONS OR SYMBOL OR SYMBOLS OR
            THUMBNAIL? OR (GUI OR GUIS OR GRAPHICAL()USER()INTERFACE?)(2N-
            )(FILE? OR ONSCREEN()REPRESENTATION? OR PICTURE? OR IMAGE?)
S2    2677128   CLICK? OR SELECT? OR DRAG? ? OR INDICAT? OR CHOOSE? OR PIC-
            K? OR INSTANTIAT? OR ACTIVAT?
S3    1408182   WRAPPER OR WRAPPERS OR WRAPS OR WRAP OR WRAPPING OR SHELL -
            OR SHELLS OR ENVELOP? OR CANVAS OR CASE? OR SHEATH?
S4       9456   S1 AND S2 AND S3
S5       1882   S1(3N)S2 AND S3
S6        268   S4 AND IC=G06F-009
S7        603   S5 AND IC=G06F
S8        809   S6 OR S7
S9        641   S8 NOT AD=19941115:19971115
S10       467   S9 NOT AD=19971115:20001115
S11       368   S10 NOT AD=20001115:20031115
S12       365   S11 NOT AD=20031115:20050722
S13       241   S12 AND S5
S14       289   S1(3N)S2(3N)S3
S15       426   S3(2N)(AUTHOR OR AUTHORING OR ASSEMBL OR LAUNCH OR SPAWN OR
             LAUNCHES OR SPAWNS OR CREATES OR LAUCHING OR CREATING OR AUT-
            HORS)
S16         0   S13 AND S15
S17        79   S7 AND S14
S18         0   S7 AND S15
S19         4   S1 AND S2 AND S15
S20         1   S1(N)S2 AND S15
S21        18   S1 AND S15
S22        83   S2 AND S15
S23       372   S17 OR S19 OR S20 OR S22 OR S14
S24       128   S23 AND IC=G06F
S25       105   S24 NOT AD=19941115:19971115
S26        74   S25 NOT AD=19971115:20001115
S27        54   S26 NOT AD=20011115:20041115
S28        54   S27 NOT AD=20041115:20051201
S29        54   IDPAT (sorted in duplicate/non-duplicate order)
S30        54   IDPAT (primary/non-duplicate records only)
File 347:JAPIO Nov 1976-2005/Feb(Updated 050606)
         (c) 2005 JPO & JAPIO
File 350:Derwent WPIX 1963-2005/UD,UM &UP=200545
         (c) 2005  Thomson Derwent
```

30/5/9 (Item 9 from file: 350)
DIALOG(R)File 350:Derwent WPIX

010571638 **Image available**
WPI Acc No: 1996-068591/199607
XRPX Acc No: N96-057708

Pre-existing shared library using multi-thread computer system manipulation method - involves acquiring write-exclusive lock for particular called library function to which call is then made and upon return from real routine, write-exclusive lock is unlocked and return to user is executed

Patent Assignee: INT BUSINESS MACHINES CORP (IBMC)
Inventor: PEEK J S
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
US 5481706	A	19960102	US 93143586	A	19931101	199607	B

Priority Applications (No Type Date): US 93143586 A 19931101
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
US 5481706	A		11	G06F-013/00	

Abstract (Basic): US 5481706 A

The method involves using a pre-existing function descriptor, its pointer, and a corresponding function which is not multithread-safe when used in the system. The system also has a corresponding underlying call associated with it which is exported from the library to render the shared library multithread-safe when used in the system. A wrapper is created for the function which has a program code extending around **instantiation** of the underlying call and which is responsive to a call corresponding to the underlying call.

The creation involves storing the pre-existing function descriptor within, and creating a lock to the function. The pre-existing shared library is then rebound with the **wrapper** . This involves **creating** a modified function descriptor by creating a pointer to the wrapper. A copy of the function descriptor is also created with the pointer to the wrapper in the modified descriptor being substituted for the function pointer in the function descriptor during the rebinding. A pointer is substituted to the modified pointer descriptor for the pointer to the function descriptor.

ADVANTAGE - Ensures correct functioning and integrity of identified library functions accessible by multiple threads. Provides multithread-safe shared libraries without necessitating extensive library source modifications.

Dwg.2/4

Title Terms: PRE; EXIST; SHARE; LIBRARY; MULTI; THREAD; COMPUTER; SYSTEM; MANIPULATE; METHOD; ACQUIRE; WRITING; EXCLUDE; LOCK; CALL; LIBRARY; FUNCTION; CALL; MADE; RETURN; REAL; ROUTINE; WRITING; EXCLUDE; LOCK; UNLOCK; RETURN; USER; EXECUTE
Derwent Class: T01
International Patent Class (Main): **G06F-013/00**
International Patent Class (Additional): **G06F-015/16**
File Segment: EPI

30/5/15 (Item 15 from file: 350)
DIALOG(R)File 350:Derwent WPIX

009479845
WPI Acc No: 1993-173380/199321
XRPX Acc No: N93-132941
Providing virtual classes for discrete objects without sub-classing - enabling user or developer to easily modify folder or any other object to gain locking or timed property
Patent Assignee: ANONYMOUS (ANON)
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
RD 348077	A	19930410	RD 93348077	A	19930320	199321	B

Priority Applications (No Type Date): RD 93348077 A 19930320
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
RD 348077	A		1	G06F-000/00	

Abstract (Basic): RD 348077 A
A Virtual Class is created called VC password. The class provides an **enveloping** mechanism which can be absorbed into existing class based objects. Virtual classes cannot exist as an **instantiation** , it must **envelope** an existing **object** or class. An **envelope** template is provided which a user can **drag** to an existing **Object** to inherit the characteristics of the Virtual Class.
To create a WP Folder Class object with a password characteristics the user can create a Folder Object using the WP Folder Class template then **drag** the VC password **envelope** template onto the **Object** . The **object** is now a password protected folder object. In addition, the user could drag the VC password template onto an existing Class Template to create a template which will produce class objects with the VC password inherited characteristic.
ADVANTAGE - Reduction of production of class names to facilitate characteristics from superclasses.
Dwg.0/0
Title Terms: VIRTUAL; CLASS; DISCRETE; OBJECT; SUB; ENABLE; USER; DEVELOP; EASY; MODIFIED; FOLDER; OBJECT; GAIN; LOCK; TIME; PROPERTIES
Derwent Class: T01
International Patent Class (Main): **G06F-000/00**
File Segment: EPI

30/5/16 (Item 16 from file: 350)
DIALOG(R)File 350:Derwent WPIX

009479809
WPI Acc No: 1993-173344/199321
XRPX Acc No: N93-132908
Direct access to contained objects via cascaded menu - allows direct selection of contained objects and grouping of objects to reduce desk-top clutter
Patent Assignee: ANONYMOUS (ANON)
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
RD 348041	A	19930410	RD 93348041	A	19930320	199321	B

Priority Applications (No Type Date): RD 93348041 A 19930320
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
RD 348041	A		1	G06F-000/00	

Abstract (Basic): RD 348041 A
The user is given an additional open option when **selecting** a folder **object** . That option would allow the user to directly open a contained object without first opening the folder. Currently, the user may choose to open the folder to a variety of views such as icons or settings.
The additional option would allow the user to choose an option such as ''open specific **object** ''. **Selection** of this choice would cause another list to appear in a cascade menu that lists the **objects** contained in that **object** , in this **case** a folder. Note that **selection** of an **object** in this list would automatically cause this list to disappear.
ADVANTAGE - Eliminates wasted user actions such as opening container objects and then closing them. Allows user to organise desktops more efficiently. Reduces user errors by reducing number of steps required.
Dwg.0/0
Title Terms: DIRECT; ACCESS; CONTAIN; OBJECT; CASCADE; MENU; ALLOW; DIRECT; SELECT; CONTAIN; OBJECT; GROUP; OBJECT; REDUCE; DESK; TOP; CLUTTER
Derwent Class: T01
International Patent Class (Main): **G06F-000/00**
File Segment: EPI

30/5/20 (Item 20 from file: 350)
DIALOG(R)File 350:Derwent WPIX

008894800
WPI Acc No: 1992-022069/199203
XRPX Acc No: N92-016734
Display item selection visual indicator - allows ICON selection **in which shrink** wrap **grey and black dithered outline appears around opaque pels of ICON**
Patent Assignee: ANONYMOUS (ANON)
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
RD 332055	A	19911210				199203	B

Priority Applications (No Type Date): RD 91332055 A 19911120

Abstract (Basic): RD 332055 A
The **selection** visual **indicator** allows **icon selection** and **icon** label editing. For **selection** , a `shrink **wrap** ' grey and black dithered outline appears around the opaque pels of the **icon** . For `in-use' **indications** , a background square of dithered light grey and yellow appears as the in-use visual.
A dashed or dotted line is incorporated around the **object** . To **select** the **icon** label for editing, the user invokes text edit for the entry area with a cursor indicating the position to type text. The `target' visual appears when an object is being dragged to another object. It is a black box around the object. It appears when the mouse button is held down, or when the drag task is still being performed.
ADVANTAGE - Conveys message neatly and cleanly without demanding too much attention, as would swatch of colour and adding capability to show `in-use' `cursor-selected' and `target' increases amount of function. (1pp Dwg.No.0/0
Title Terms: DISPLAY; ITEM; SELECT; VISUAL; INDICATE; ALLOW; SELECT; SHRINK ; **WRAP** ; GREY; BLACK; DITHER; OUTLINE; APPEAR; OPAQUE
Derwent Class: T01
International Patent Class (Additional): **G06F-000/01**
File Segment: EPI

30/5/31 (Item 31 from file: 347)
DIALOG(R)File 347:JAPIO

04731840 **Image available**
DYNAMIC ICON PREPARING DEVICE

PUB. NO.: 06-202840 [JP 6202840 A]
PUBLISHED: July 22, 1994 (19940722)
INVENTOR(s): JINBA TOMONARI
APPLICANT(s): NEC CORP [000423] (A Japanese Company or Corporation), JP (Japan)
APPL. NO.: 04-341743 [JP 92341743]
FILED: December 22, 1992 (19921222)
INTL CLASS: [5] **G06F-003/14**
JAPIO CLASS: 45.3 (INFORMATION PROCESSING -- Input Output Units)
JAPIO KEYWORD: R002 (LASERS); R004 (PLASMA); R011 (LIQUID CRYSTALS); R101 (APPLIED ELECTRONICS -- Video Tape Recorders, VTR); R102 (APPLIED ELECTRONICS -- Video Disk Recorders, VDR); R131 (INFORMATION PROCESSING -- Microcomputers & Microprocessers)

ABSTRACT

PURPOSE: To save labor for dynamic icon preparation by storing the data of the respective patterns of dynamic icons and setting the functions of the dynamic icons.

CONSTITUTION: When a user designates video fetch, a video fetching means 104 transfers the output signal of a video storage means 102 to a data storage means 105 as it is or while converting data. This purpose is to extract only the video utilized for the dynamic icon and to preserve it in the data storage means 105 among the video stored in the video storage means 102. When the user instructs function setting, a function setting means 106 preserves the function designated by the user in a function storage means 107. In this **case** , the set function is a function to be executed when the user **selects** the dynamic **icon** in the **case** of utilizing the prepared dynamic **icon** while integrating it into the other system.

30/5/37 (Item 37 from file: 347)
DIALOG(R)File 347:JAPIO

04205806 **Image available**
METHOD FOR INDICATING DATA INPUT AND OUTPUT OF WINDOW SYSTEM

PUB. NO.: 05-197506 [JP 5197506 A]
PUBLISHED: August 06, 1993 (19930806)
INVENTOR(s): SHIGEGAKI MASATO
HASEGAWA TAKASHI
KITAHARA YOSHINORI
APPLICANT(s): HITACHI LTD [000510] (A Japanese Company or Corporation), JP (Japan)
APPL. NO.: 04-008210 [JP 928210]
FILED: January 21, 1992 (19920121)
INTL CLASS: [5] **G06F-003/14** ; **G06F-003/033** ; **G06F-015/62**
JAPIO CLASS: 45.3 (INFORMATION PROCESSING -- Input Output Units); 45.4 (INFORMATION PROCESSING -- Computer Applications)
JOURNAL: Section: P, Section No. 1645, Vol. 17, No. 621, Pg. 159, November 16, 1993 (19931116)

ABSTRACT

PURPOSE: To indicates input and output between files in interactive and reusable form on the window system by storing a file with the coupling relation between application programs which have no data input/output indication screen.

CONSTITUTION: For example, when a spelling check is performed by applying desk-top publishing wherein a text editor, a spelling checker, typesetting software, a previewer, and a printer are combined, the application programs are combined on an input/output **indication** screen 201. In this **case** , a text **icon** and application **icons** on a file control screen are moved to the input/output control screen 201 and connected by coupling lines, and application software for processing and a bulb icon for input/output switching are placed between them to complete the combination. Further, the text 202 which is currently edited is connected to the spelling checker 206 through the bulb icon 204 and the output of the spelling checker 206 is connected to the text previewer 209.

30/5/48 (Item 48 from file: 347)
DIALOG(R)File 347:JAPIO

03535264 **Image available**
OBJECTIVE ELEMENT DESIGNATING SYSTEM IN INTERACTIVE GRAPHIC PROCESSING

PUB. NO.: 03-198164 [JP 3198164 A]
PUBLISHED: August 29, 1991 (19910829)
INVENTOR(s): UEDA TATSUYA
APPLICANT(s): FUJITSU LTD [000522] (A Japanese Company or Corporation), JP (Japan)
APPL. NO.: 01-341325 [JP 89341325]
FILED: December 27, 1989 (19891227)
INTL CLASS: [5] **G06F-015/62**
JAPIO CLASS: 45.4 (INFORMATION PROCESSING -- Computer Applications)
JOURNAL: Section: P, Section No. 1280, Vol. 15, No. 466, Pg. 48, November 26, 1991 (19911126)

ABSTRACT

PURPOSE: To simplify the operation for designating a **selection object** by batch- designating plural graphic elements which are displayed, and thereafter, designating a non- **selection object** in them and determining an objective graphic.

CONSTITUTION: First of all, plural graphic element containing a **selection object** are batch-designated by a first graphic element designating means 121. In this **case** , discriminating information corresponding to each of plural graphic elements which are designated is stored in an objective graphic information storing means 131. Subsequently, by a second graphic element designating means 123, that which becomes a non- **selection object** in plural graphic elements which are batch-designated is designated. In an object graphic determining means 141, by storing the information of a fact that the graphic element designated thereby is excluded from the **selection object** in the objective graphic information store means 131, the graphic element excluding that which becomes a non- **selection object** in plural graphic elements batch-designated is determined as the **selection object** . In such a way, in the **case** the **selection object** graphic and the non- **selection object** graphic are mixed and displayed, the **selection object** is easily designated.

01283450 ORDER NO: NOT AVAILABLE FROM UNIVERSITY MICROFILMS INT'L.

A UNIFIED COMPUTATIONAL MODEL FOR SCHEMAS AND NEURAL NETWORKS IN CONCURRENT OBJECT-ORIENTED PROGRAMMING (OBJECT-ORIENTED PROGRAMMING, PROGRAMMING)

Author: WEITZENFELD, ALFREDO JOSE
Degree: PH.D.
Year: 1992
Corporate Source/Institution: UNIVERSITY OF SOUTHERN CALIFORNIA (0208)
Chair: MICHAEL ARBIB
Source: VOLUME 53/12-B OF DISSERTATION ABSTRACTS INTERNATIONAL.
PAGE 6406.
Descriptors: COMPUTER SCIENCE
Descriptor Codes: 0984

This dissertation presents, in Part I, the Abstract Schema Language (ASL), unifying schema modeling with concurrent object-oriented programming (COOP). ASL extends the current state of the art in both areas by providing a hierarchical approach towards heterogeneous and multi-granular concurrent object design. Schemas in ASL are functional units which get implemented in an orthogonal fashion. This aspect not only encourages **code** reusability, but enables schema implementations as, e.g., procedural programs or neural network processes. ASL addresses the need to separate task specification from actual task implementation, by utilizing the concept of dynamic task delegation and static **wrapping** of external programs. ASL schemas define class templates from which active **objects** get dynamically **instantiated** . Schemas incorporate a dynamic interface made of multiple input and output ports, plus a body section which explicitly specifies the schema behavior. Communication is in the form of asynchronous message passing, hierarchically managed through anonymous port reading and writing, at one level, and dynamic port inter-connections and relabelings, specified at a higher level. ASL supports object composition, through the notion of schema assemblages, enabling top-down and bottom-up model designs. The semantics of ASL are described in terms of a Structural Operational Semantics (SOS), complemented by the machine implementation description. A comparison is given between ASL and other schema and concurrent object-based systems.

Part II of the thesis describes the Neural Schema Language (NSL), developed as a domain specific system based on the ASL model. NSL supports the development of complex applications in the areas of Brain Theory and Distributed Artificial Intelligence. The NSL model supports modeling of neurons, described with different levels of internal detail, and neural networks which may be inter-connected as networks of networks. Domain specific examples are given in NSL. The system is then compared to other neural networks simulation systems.

Finally, future research paths are described for both ASL and NSL. (Copies available exclusively from Micrographics Department, Doheny Library, USC, Los Angeles, CA 90089-0182.)

30/5/15 (Item 15 from file: 350)
DIALOG(R)File 350:Derwent WPIX

009479845
WPI Acc No: 1993-173380/199321
XRPX Acc No: N93-132941
Providing virtual classes for discrete objects without sub-classing - enabling user or developer to easily modify folder or any other object to gain locking or timed property
Patent Assignee: ANONYMOUS (ANON)
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
RD 348077	A	19930410	RD 93348077	A	19930320	199321	B

Priority Applications (No Type Date): RD 93348077 A 19930320
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
RD 348077	A		1	G06F-000/00	

Abstract (Basic): RD 348077 A

A Virtual Class is created called VC password. The class provides an **enveloping** mechanism which can be absorbed into existing class based objects. Virtual classes cannot exist as an **instantiation** , it must **envelope** an existing **object** or class. An **envelope** template is provided which a user can **drag** to an existing **Object** to inherit the characteristics of the Virtual Class.

To create a WP Folder Class object with a password characteristics the user can create a Folder Object using the WP Folder Class template then **drag** the VC password **envelope** template onto the **Object** . The **object** is now a password protected folder object. In addition, the user could drag the VC password template onto an existing Class Template to create a template which will produce class objects with the VC password inherited characteristic.

ADVANTAGE - Reduction of production of class names to facilitate characteristics from superclasses.

Dwg.0/0

Title Terms: VIRTUAL; CLASS; DISCRETE; OBJECT; SUB; ENABLE; USER; DEVELOP; EASY; MODIFIED; FOLDER; OBJECT; GAIN; LOCK; TIME; PROPERTIES
Derwent Class: T01
International Patent Class (Main): **G06F-000/00**
File Segment: EPI

30/5/16 (Item 16 from file: 350)
DIALOG(R)File 350:Derwent WPIX
(c) 2005 Thomson Derwent. All rts. reserv.

009479809
WPI Acc No: 1993-173344/199321
XRPX Acc No: N93-132908
Direct access to contained objects via cascaded menu - allows direct selection of contained objects and grouping of objects to reduce desk-top clutter
Patent Assignee: ANONYMOUS (ANON)
Number of Countries: 001 Number of Patents: 001
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week	
RD 348041	A	19930410	RD 93348041	A	19930320	199321	B

Priority Applications (No Type Date): RD 93348041 A 19930320
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
RD 348041	A		1	G06F-000/00	

Abstract (Basic): RD 348041 A

The user is given an additional open option when **selecting** a folder **object** . That option would allow the user to directly open a contained object without first opening the folder. Currently, the user may choose to open the folder to a variety of views such as icons or settings.

The additional option would allow the user to choose an option such as ''open specific **object** ''. **Selection** of this choice would cause another list to appear in a cascade menu that lists the **objects** contained in that **object** , in this **case** a folder. Note that **selection** of an **object** in this list would automatically cause this list to disappear.

ADVANTAGE - Eliminates wasted user actions such as opening container objects and then closing them. Allows user to organise desktops more efficiently. Reduces user errors by reducing number of steps required.

Dwg.0/0

Title Terms: DIRECT; ACCESS; CONTAIN; OBJECT; CASCADE; MENU; ALLOW; DIRECT; SELECT; CONTAIN; OBJECT; GROUP; OBJECT; REDUCE; DESK; TOP; CLUTTER
Derwent Class: T01
International Patent Class (Main): **G06F-000/00**
File Segment: EPI